LINUSE: Libraries IN USEr space

En serio, no estoy tratando de destruir al Kernel. Ése será tan sólo un efecto colateral no intencionado.

Cátedra de Sistemas Operativos

Trabajo práctico Cuatrimestral

2C2019

Historial de versiones

v1.0 (05/09/2019) Publicación Inicial

v1.1 (08/10/2019) Primera Revisión

- *Aclaramos que tanto MUSE como SAC-Server deben ser capaces de atender pedidos de distintos hilos en paralelo*
- *Aclaramos que los marcos se recorren siempre de menor a mayor en función del número de marco*
- *Aclaramos que las direcciones lógicas son de 32 bits e indican el offset dentro de la porción de memoria que tiene asignada un hilo*
- *Aclaramos que la función init de libMUSE se debe realizar a nivel programa y no a nivel hilo*
- *Se agrega un anexo que explica el funcionamiento de la segmentación paginada en este TP*
- *Agregamos una función suse_return para poder enviar a cola de Exit hilos finalizados*
- *Se agrega la v1 de Hilolay*

v1.2 Segunda revisión

- *Aclaramos que la métrica que imprime el grado de multiprogramación es global, para todos los programas.*
- *Ampliamos un poco la información sobre hilolay del enunciado para que iguale al repo de Hilolay.*
- *Se agrega la v2 de Hilolay, ahora con semáforos!*

v1.3 Tercer revisión

- *Arreglamos un error en el anexo de memoria que día que solo se deben buscar el free en el último header de muse_alloc*
- *Agregamos una aclaración sobre que la tabla de segmentos es por programa*
- *Agregamos un anexo sobre el funcionamiento en detalle del directorio de SAC-FS*

Objetivos y Normas de resolución

Objetivos del Trabajo Práctico

Mediante la realización de este trabajo se espera que el alumno:

- Adquiera conceptos prácticos del uso de las distintas herramientas de programación e interfaces (APIs) que brindan los sistemas operativos.
- Entienda aspectos del diseño de un sistema operativo.
- Afirme diversos conceptos teóricos de la materia mediante la implementación práctica de algunos de ellos.
- Se familiarice con técnicas de programación de sistemas, como el empleo de makefiles, archivos de configuración y archivos de log.
- Conozca con grado de detalle la operatoria de Linux mediante la utilización de un lenguaje de programación de relativamente bajo nivel como C.

Características

- Modalidad: grupal (5 integrantes +- 0) y obligatorio

- Tiempo estimado para su desarrollo: 85 días
- Fecha de comienzo: 07/09/2019
- Fecha de primera entrega: 30/11/2019
- Primer Recuperatorio: 07/12/2019
- Segundo Recuperatorio: 21/12/2019
- Lugar de corrección: Laboratorio de sistemas, sede Medrano

Evaluación del Trabajo Práctico

El trabajo práctico consta de una evaluación en 2 etapas.

La primera etapa consistirá en evaluar funcionalmente los programas desarrollados por el grupo, de forma presencial en el laboratorio. Para ello, se llevarán adelante ciertas "pruebas" (tests) las cuales serán provistas con suficiente tiempo para que los alumnos puedan evaluarlas con antelación. Queda aclarado que para que un trabajo práctico sea considerado evaluable, el mismo debe proporcionar **registros de su funcionamiento de la forma más clara posible**. Esto es, que pueda entenderse a la perfección todos los estados y estructuras del trabajo práctico, desde su inicialización, hasta su apagado.

La segunda etapa se dará en caso de aprobada la primera y constará de un coloquio, con el objetivo de afianzar los conocimientos adquiridos durante el desarrollo del trabajo práctico y terminar de definir la nota de cada uno de los integrantes del grupo. Por lo tanto se recomienda que cada grupo abogue por obtener una distribución de carga de trabajo equitativa, que permita el aprendizaje de todos los miembros del grupo.

Cabe aclarar que el trabajo equitativo no asegura la aprobación de la totalidad de los integrantes, sino que cada uno tendrá que defender y explicar tanto teórica como prácticamente los conceptos desarrollados y aprendidos a lo largo de la cursada.

La defensa del trabajo práctico (o "coloquio") consta de la relación de lo visto durante la teoría con lo implementado. De esta manera, una implementación que contradiga a los contenidos vistos en las clases teóricas que no esté explícitamente dado en este documento, ***será motivo de desaprobación del trabajo práctico***.

Deployment y Testing del Trabajo Práctico

Al tratarse de una plataforma distribuida, los procesos involucrados podrán ser ejecutados en diversas computadoras. La cantidad de computadoras involucradas y la distribución de los diversos procesos en las mismas será definida en cada uno de los tests y **siendo posible que la misma se modifique en el momento de la evaluación**. Es responsabilidad del grupo automatizar el despliegue de los diversos procesos con sus correspondientes archivos de configuración para cada uno de los diversos tests a evaluar.

Todo esto estará detallado en el documento de pruebas que se publicará cercano a la fecha de Entrega Final. Archivos y programas de ejemplo se pueden encontrar en el repositorio de la cátedra.

Finalmente, recordar la existencia de las Normas del Trabajo Práctico[1] donde se especifican todos los lineamientos de cómo se desarrollará la materia durante el cuatrimestre.

Aclaraciones

Debido al fin académico del trabajo práctico, los conceptos reflejados son, en general, versiones simplificadas o alteradas de los componentes reales de hardware y de sistemas operativos modernos, a fin de resaltar aspectos de diseño.

Invitamos a los alumnos a leer las notas y comentarios al respecto que haya en el enunciado, reflexionar y discutir con sus compañeros, ayudantes y docentes al respecto.

Introducción

El objetivo de este trabajo práctico es desarrollar varios módulos que nos van a permitir **proveer servicios similares** a los que expone el sistema operativo. En particular:

1. Creación y gestión de hilos de ejecución (Bajo las interfaces de **Hilolay** y **SUSE**).
2. Gestión de Memoria Dinámica (Bajo las interfaces de **libMUSE** y **MUSE**).
3. Gestión de un Sistema de Archivos (Bajo las interfaces de **FUSE** y **SAC**).

Esto nos va a permitir **desarrollar programas en C que utilice nuestras funciones y que las mismas resulten equivalentes a utilizar las del sistema operativo**. De esa misma manera, en las instancias de prueba se va a evaluar el trabajo práctico con programas hechos para demostrar todas las funcionalidades.

¿Cómo podemos reemplazar las funcionalidades del Sistema Operativo?

Por lo general, los Sistemas Operativos proveen servicios a los programas a través de funciones, que resuelven diversas necesidades desde abrir un archivo hasta crear un nuevo proceso. Dichas funciones, llamadas "*syscalls*" (o "llamadas al sistema"), se suelen agrupar cohesivamente en paquetes llamados "**bibliotecas**" *-o libraries en inglés-*. Estas no están limitadas a ser solo del sistema operativo, sino que **cada programador puede crear las propias**[2].

De esa misma manera, en vez de usar las funciones del sistema operativo, vamos a usar las que desarrollemos en el trabajo práctico:

```
#include <stdlib.h>                    ←→   #include "libmuse.h"

void main(){                                void main(){
  void* my_memory = malloc(10);        ←→     uint32_t my_memory = muse_alloc(10);
  ...                                           ...
  free(my_memory);                     ←→     muse_free(my_memory);
}                                           }
```

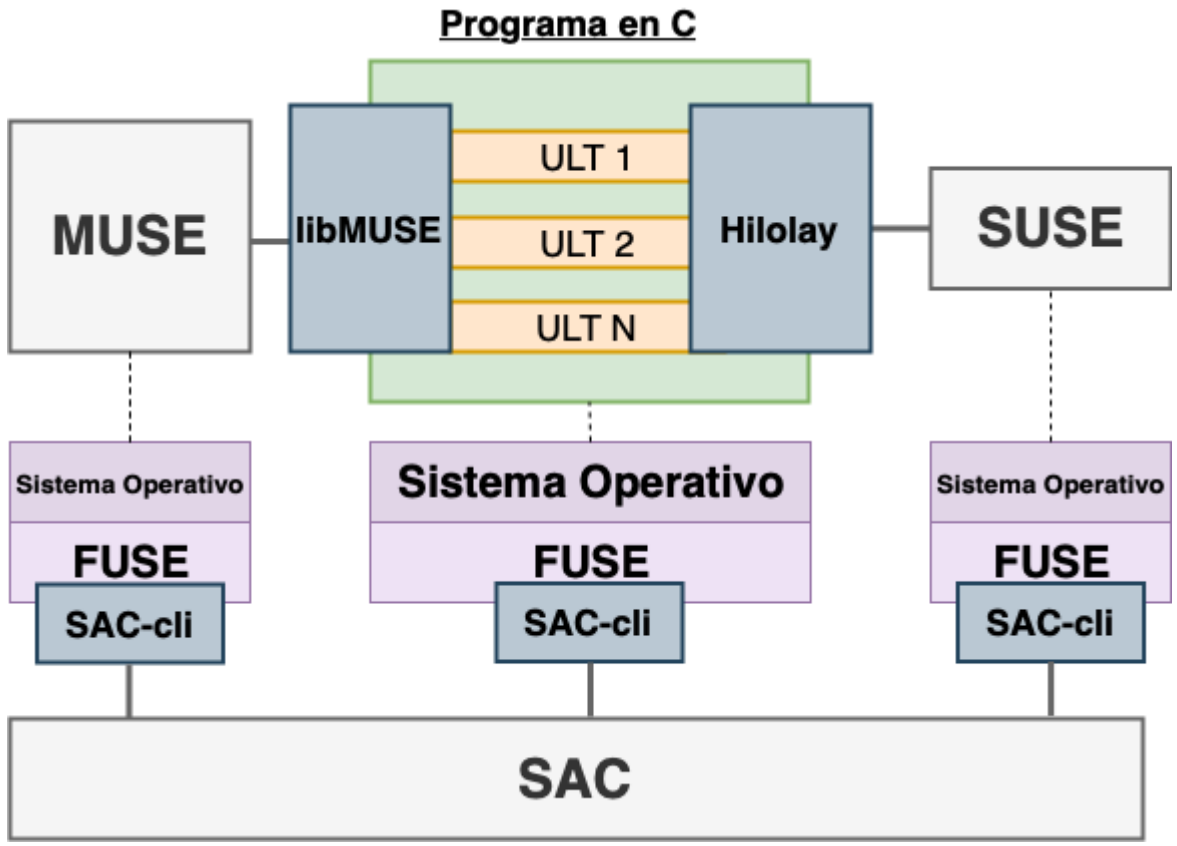
[1] https://faq.utnso.com/ntp
[2] Por ejemplo, en la cátedra tenemos desarrolladas las commons: sisoputnfrba/so-commons-library, que les van a ser bastante útiles en el desarrollo del TP.

Estas bibliotecas son **software que tiene entidad propia**. Eso significa que deberán ser compiladas y publicadas por separado[3], para poder ser utilizadas para compilar un programa como el anterior.

Diagrama de Arquitectura

La mayoría de los módulos van a exponer bibliotecas para interactuar con nuestros programas. Esas bibliotecas van a actuar como **sub-módulos**, comunicándose con su contraparte **centralizada**, en diferentes computadoras.

Si agrupamos todos estos submódulos en un único diagrama, obtendremos lo siguiente:



Nota: FUSE es una implementación que vamos a utilizar para evitar usar bibliotecas para gestionar nuestro Sistema de Archivos. Más adelante ampliaremos su funcionamiento e integración con nuestro módulo.

[3] En este video pueden ver como se puede realizar la compilación de una biblioteca con Eclipse.

Módulo de Planificación

Este módulo nos permitirá la **creación y gestión de ULTs**[4], planificándolos de forma **centralizada**. Su interacción con los programas será análoga a otras bibliotecas de gestión de hilos, como lo es pthread:

```
#include <stdio.h>                          ↔  #include <stdio.h>
#include <pthread.h>                        ↔  #include "hilolay.h"

void main(){                                ↔  void main(){
                                            ↔    hilolay_init();
  int x = 0;                                ↔    int x = 0;
  pthread_t thread;                         ↔    hilolay_t thread;
  pthread_create(&thread, NULL, fun, &x);   ↔    hilolay_create(&thread, NULL, fun, &x);
  ...                                       ↔    ...
  pthread_join(thread, NULL);               ↔    hilolay_join(thread, NULL);
}                                           ↔  }

int fun(int* x){                            ↔  int fun(int* x){
  int y = 0;                                ↔    int y = 0;
                                            ↔    hilolay_yield();
  y = 2;                                    ↔    y = 2;
  return 0;                                 ↔    return 0;
}                                           ↔  }
```

De esa manera, nos va a permitir utilizar nuestro propio sistema de gestión de hilos sin demasiados cambios en el código.

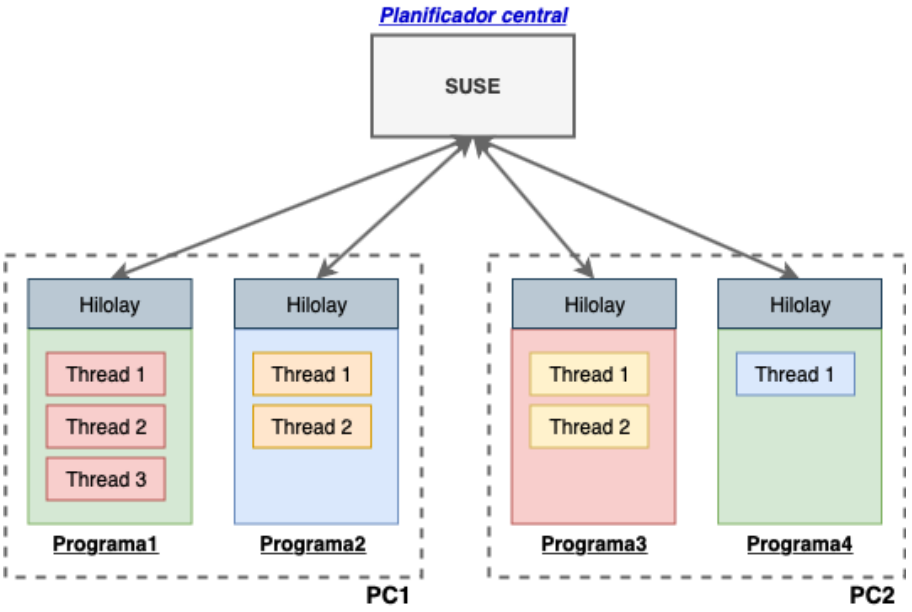
Este módulo va a estar dividido en dos **sub-módulos**, que se comunicarán mediante sockets TCP/IP:

- La biblioteca que vemos arriba: **Hilolay**.
- El sistema de planificación centralizado: **SUSE**.

Para simplificar la tarea de desarrollo, **la parte de Hilolay que interactúa con los programas ya se encuentra desarrollada por la cátedra**[5], por lo que los grupos solo necesitarán implementar una serie de funciones para poder comunicarlo con su implementación de SUSE.

Debido a que los programas que utilicen la biblioteca Hilolay no necesariamente estarán corriendo en las mismas computadoras, su interacción con SUSE se puede ver de esta forma:

[4] *Página 165*, Capítulo 4 del libro "Sistemas operativos. Aspectos internos y principios de diseño", William Stallings, 5ta Edición.
[5] Repositorio donde se encuentra el desarrollo: https://github.com/sisoputnfrba/hilolay



Biblioteca Hilolay

Tal como mencionamos anteriormente, Hilolay va a necesitar que cada grupo implemente las siguientes funciones internas de la misma, donde se deberá interactuar con SUSE:

hilolay_init: Esta función es la encargada de inicializar todos los recursos de la biblioteca.

suse_create: Esta función es invocada cuando se necesita crear un nuevo hilo, donde la función que se pase por parámetro actuará como main del mismo, finalizando el hilo al terminar esa función.

suse_schedule_next: Obtiene el próximo hilo a ejecutar.

suse_wait: Genera una operación de wait sobre el semáforo dado. Retorna si el hilo pudo obtener el semáforo.[6]

suse_signal: Genera una operación de signal sobre el semáforo dado.

suse_join: Bloquea al thread esperando que el mismo termine. El thread actual pasará a estar BLOCKED y saldrá del mismo luego de que el TID indicado finalice su ejecución. También es posible realizar un join a un thread ya finalizado.

suse_close: Da por finalizado al TID indicado. El thread actual pasará a estar EXIT.

Scheduler in USEr space (SUSE)

SUSE es un **planificador centralizado en espacio de usuario** de cada programa que use la biblioteca Hilolay.

Si bien los hilos de un programa serán independientes de los de cualquier otro en término de la *planificación*, podrán compartir recursos entre ellos por su propia naturaleza (tanto entre hilos del

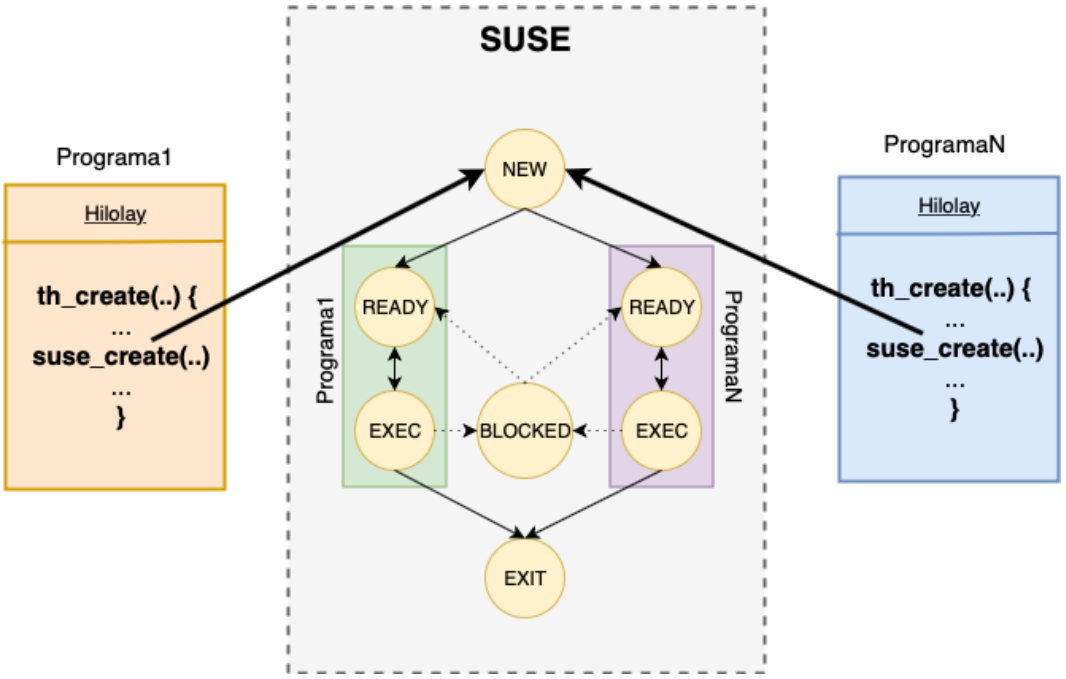
[6] El comportamiento es análogo a su contraparte estándar, sem_wait. Pueden revisarlo haciendo man sem_wait.

mismo o de distinto programa), pudiendo provocar una *condición de carrera*[7]. Para solucionar eso, el módulo tiene su propia implementación de **semáforos**.

Además, SUSE deberá saber en **todo momento los procesos que fueron planificados** en él, así como las **métricas** de los mismos y de sus ejecuciones.

Diagrama de estados

SUSE mantendrá al mismo tiempo una planificación a corto y a largo plazo. En su sistema de planificación a largo plazo utilizará un algoritmo FIFO, mientras que a corto plazo utilizará **Shortest Job First**[8] (SJF), en su implementación estadística, con estimación inicial 0. SUSE mantendrá un diagrama de estados similar al que es visto en la parte teórica de la materia:



Comparándolo con el diagrama tradicional, hay que tener en cuenta:

1. Se mantendrá un estado BLOCKED común a todos los hilos de todos los programas.

Un hilo puede transicionar a este estado mediante un bloqueo por **semáforo o esperando que termine otro thread**. Así mismo, cuando se libere el recurso que lo bloqueó, el hilo pasará a estado READY del programa al que pertenece. En caso de existir más de un hilo a transicionar, *quedará a disposición de cada grupo definir el algoritmo de selección para la cola del semáforo*.

2. El estado EXIT también será común a todos los hilos de todos los programas.

3. El estado NEW será común a todos los hilos de todos los programas.

[7]Ver el Capítulo 6 (6.1): Fundamentos de Sincronización del libro "Fundamentos de Sistemas Operativos", Silbertchatz, Galvin, Gagne 7ma Edición.
[8] Capítulo 5 (5.3.2): **Algoritmos de Planificación** del "Fundamentos de Sistemas Operativos", Silbertchatz 7ºEd.

Las transiciones de los hilos a READY serán según el **grado de multiprogramación**. El mismo será único y será compartido por todos los hilos. En caso de que el grado de multiprogramación no permita que el hilo ingrese inmediatamente, el *mismo quedará encolado hasta que le sea posible, siendo esto transparente para el proceso que lo creó*.

4. Los estados READY y EXEC son locales a cada uno de los programas.

Cada programa representará una CPU *virtual*. Eso implica que solo existirá un único ULT en estado EXEC por programa.

Recursos Compartidos: Semáforos

En orden con las funciones de semáforos de Hilolay, vamos a necesitar que SUSE tenga su propia implementación de **semáforos**[9][10]. Estos van a estar definidos con un identificador alfanumérico y un valor inicial dados *por archivo de configuración*. Estos semáforos podrán ser tanto semáforos contadores, como semáforos binarios.

Métricas

Como se mencionó anteriormente, una de las responsabilidades de SUSE es la de administrar los hilos en ejecución y poder comunicar **métricas** sobre los mismos que permitan dar un seguimiento a sus planificaciones. Las métricas deben ser escritas en un archivo de log bajo dos circunstancias:

- Cada cierto intervalo definido por archivo de configuración
- Cuando un hilo finaliza

Las métricas mínimas (pudiendo el grupo agregar otras) son las siguientes:

- **Por cada Hilo:**
  - **Tiempo de ejecución** (intervalo en milisegundos desde la creación hasta el momento de tomar la métrica)
  - **Tiempo de espera** (suma de los intervalos en milisegundos en estado Ready)
  - **Tiempo de uso de CPU** (suma de los intervalos en milisegundos en estado Exec)
  - **Porcentaje del Tiempo de Ejecución** (tiempo de ejecución del hilo dividido la suma del tiempo de ejecución de todos los hilos de ese proceso).
- **Por cada programa:**
  - Cantidad de hilos en cada estado (NEW, READY, RUN, BLOCKED)
- **Del sistema**
  - Por cada semáforo, su valor actual
  - Grado actual de multiprogramación

Archivo de configuración[11]

Campo	Tipo	Descripción
LISTEN_PORT	[numérico]	Puerto TCP utilizado para recibir las conexiones de CPU y I/O

[9] Capítulo 6.5: **Semáforos** del libro “Fundamentos de Sistemas Operativos”, Silbertchatz 7ma Edición
[10] Test and Set instructions: https://en.wikipedia.org/wiki/Test-and-set
[11] Queda a decisión del grupo el agregado de más parámetros al mismo.

METRICS_TIMER	[numérico]	Intervalo de tiempo en segundos entre cada loggeo de métricas
MAX_MULTIPROG	[numérico]	Grado máximo de Multiprogramación del Sistema
SEM_IDS	[array]	Identificador alfanumérico de cada semáforo del sistema. Cada posición del array representa un semáforo
SEM_INIT	[array]	Valor inicial de cada semáforo definido en SEM_IDS, según su posición
SEM_MAX	[array]	Valor máximo de cada semáforo definido en SEM_IDS, según su posición.
ALPHA_SJF	[numérico]	Parámetro de peso relativo del algoritmo SJF

Ejemplo de Archivo de Configuración

```
LISTEN_PORT=5003
METRICS_TIMER=50
MAX_MULTIPROG=10
SEM_IDS=[A, B]
SEM_INIT=[0, 1]
SEM_MAX=[1, 5]
ALPHA_SJF=0.5
```

Módulo de Memoria

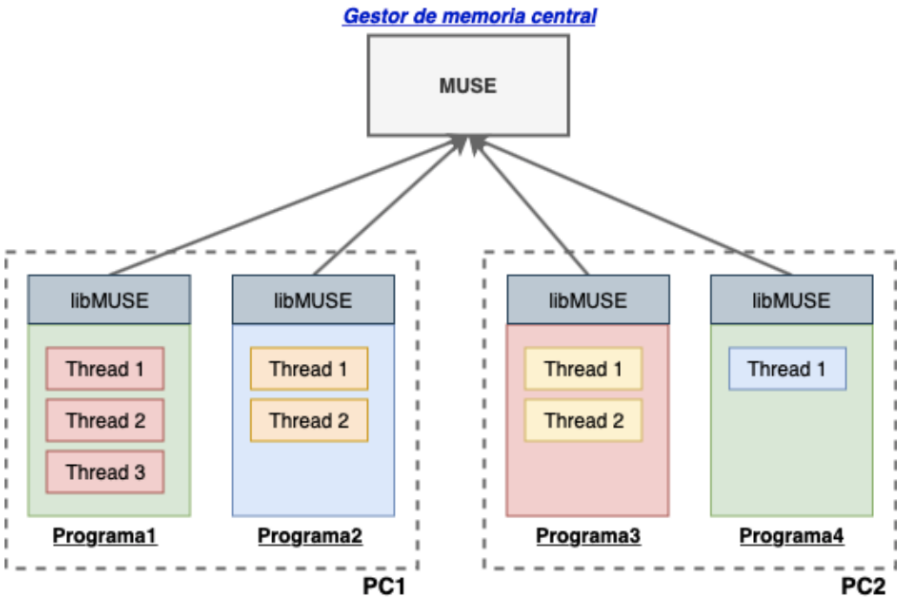
Este módulo nos permitirá **reservar y usar memoria dinámica**, almacenada de forma **centralizada**. Su interacción con los programas, va a ser análoga al conjunto de funciones de memoria dinámica de Linux, como son malloc, free, memcpy[12], etc:

```
#include <stdlib.h>                 ←   #include “libmuse.h”
#include <stdio.h>                  →   #include <stdio.h>
#include <string.h>                     #include <string.h>

void main(){                            void main(){
  void* my_memory = malloc(10);           uint32_t my_memory = muse_alloc(10);
  int x = 10;                       ←     int x = 10;
  int*y = malloc(4);                →     int*y = malloc(4);
  memcpy(my_memory, &x, 4);               muse_cpy(my_memory, &x, 4);
  *y = *((int*) my_memory);               muse_get(y, my_memory, 4);
  printf("%d", *y);                 ←     printf("%d", *y);
  ...                               →     ...
  free(my_memory);                  ←     muse_free(my_memory);
}                                   →   }

                                    ←
                                    →
```

Este módulo va a estar dividido en dos sub-módulos que se comunicarán mediante sockets TCP/IP:

- La biblioteca que vemos arriba: **libMuse**.
- El sistema de administración de memoria centralizado: **MUSE**.

[12] Para más info, ejecutar `man malloc`, `man free` o `man memcpy` en la consola de Linux.



libMUSE

libMUSE será el medio para que cada proceso se comunique con MUSE. Si bien es una biblioteca, a diferencia de Hilolay solo va a poseer una **interfaz bien definida**[13] en un repositorio provisto por la cátedra: https://github.com/sisoputnfrba/libmuse. Será responsabilidad de cada grupo el desarrollo de la totalidad de las funcionalidades subyacentes.

Al igual que sucede en la realidad, si se tratara de acceder una dirección inválida, libMUSE deberá generar una **segmentation fault**, enviando la señal SIGSEGV al proceso[14].

Aclaración: la función que realiza el init de libMUSE debe estar llamada una única vez por programa y al inicio del mismo, no se debe realizar para cada uno de los hilos.

MUSE: Memory in USEr space

MUSE será nuestro administrador centralizado de la memoria dentro del sistema. Se encargará de proveer a los programas en ejecución porciones de memoria continua, que podrán ser solicitadas y liberadas en tiempo de ejecución (estrategia normalmente conocida como “memoria dinámica”).

MUSE debe poder gestionar las distintas operaciones de los múltiples programas/hilos en paralelo.

Dichas porciones serán regiones pertenecientes a otra única porción central de memoria (UPCM), ubicada en espacio de usuario del proceso MUSE, y reservada inicialmente de forma convencional por el mismo.

Dicha porción central de memoria, se gestionará mediante varios tipos de metodologías análogas a la teoría. Su tamaño se proveerá por archivo de configuración.

[13] Ver el detalle de la interfaz en https://github.com/sisoputnfrba/libmuse/blob/master/src/libmuse.h
[14] Investigar las funciones signal y raise en C

Métricas

Para conocer el estado del sistema y poder monitorear las distintas operaciones que fueron realizadas en el sistema MUSE deberá poder comunicar **métricas** sobre cada programa. Las métricas deben ser escritas en un archivo de log bajo dos circunstancias:

- Cuando un programa finaliza, tanto correctamente como al generar un segmentation fault
- Cuando se realiza una nueva petición de memoria

Las métricas mínimas (pudiendo el grupo agregar otras) son las siguientes:

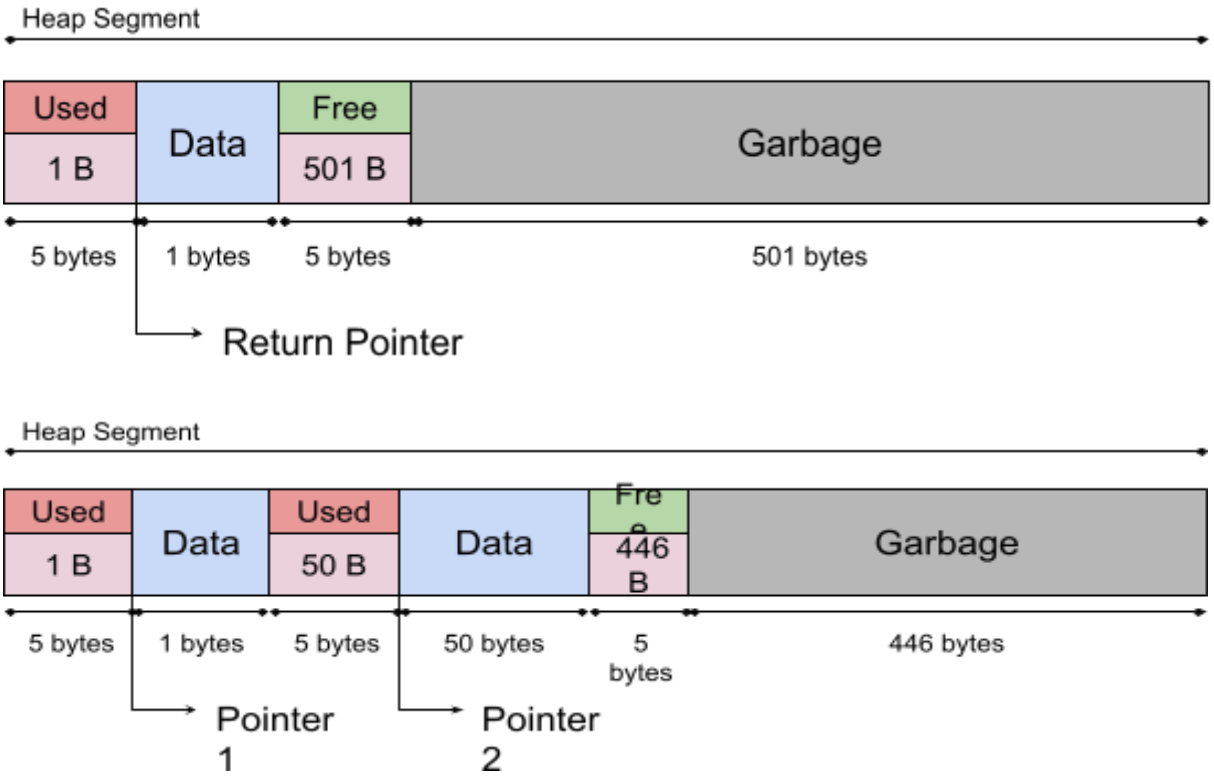
- **Por socket conectado:**
    - Porcentaje de asignación de memoria: cantidad de segmentos asignados dividido segmentos totales
    - Cantidad de memoria disponible (en bytes) en el último segmento de memoria dinámica pedido
- **Por cada programa:**
    - Memoria total pedida
    - Memoria total liberada
    - Memory leaks totales
- **Del sistema:**
    - Cantidad de memoria disponible (en bytes)

Asignación de Memoria

La asignación de memoria de este trabajo práctico utilizará el método de **segmentación paginada**. Eso implica que cada segmento estará compuesto por una o más páginas, administrando la memoria pedida mediante tanto **tablas de segmentos**, como **tablas de páginas**. De la misma forma que sucede en la MMU, *el tamaño de los segmentos estará dictado por la cantidad de páginas que lo componen.*

Cada programa, al pedir una porción de memoria, **creará un segmento** donde asignar dicha porción de memoria, entendiendo como porción **a cualquier pedido de asignación de memoria (malloc)**. Las siguientes porciones de memoria requeridas, deberán ser asignadas dentro del mismo segmento siempre y cuando sea posible.

Para distinguir dentro de un segmento cuales porciones de memoria se encuentran libres, se guardará **metadata dentro del mismo segmento**, que indicará si el bloque está vacío y cuanto tamaño tiene. Veamos un ejemplo utilizando esta metadata:



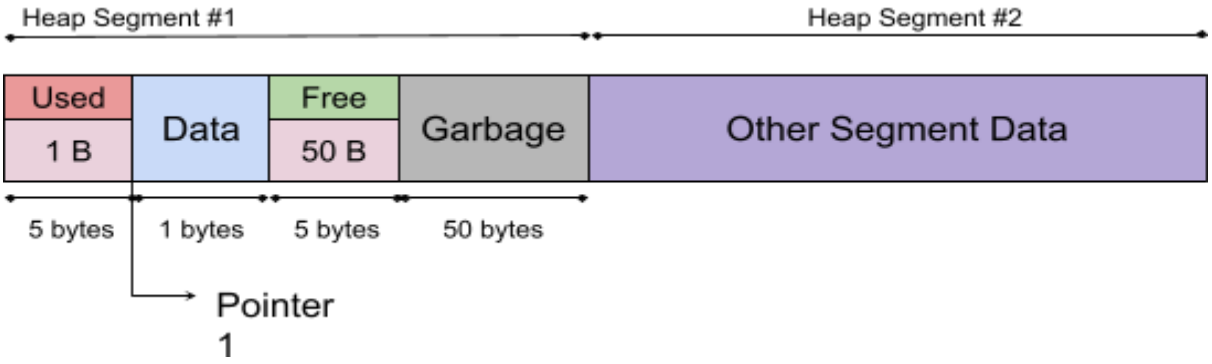
Programáticamente, esa **estructura de metadata** debería responder a los siguientes tipos:

```
typedef struct HeapMetadata {
     uint32_t size,
     Bool isFree
     }
```

Si al necesitar asignar una nueva porción de memoria a un segmento, no tiene espacio para asignar, se deberá realizar un **cambio de tamaño del segmento**. Nuestro sistema utiliza segmentación paginada, por lo que los segmentos, al siempre poseer un tamaño múltiplo del tamaño de las páginas, deberán *extender su tamaño en la cantidad de páginas que sean necesarias*.

De forma análoga, el segmento deberá achicarse si ha liberado parte de su contenido y su última página queda completamente vacía luego de una liberación de una porción de memoria, *des-asignando* la misma.

En caso de que el **segmento no pueda extenderse más** (porque, por ejemplo, pisaría una dirección virtual de otro segmento)**, se deberá crear un segmento nuevo**. Será *responsabilidad de cada grupo* elegir la **disposición que le dará a los segmentos en el espacio de direcciones**. Ese sería el caso de, por ejemplo, querer asignar una porción de memoria de 60 bytes en el Heap Segment #1:



Finalmente, como un detalle, podemos observar en la imagen, que **el valor a retornar por MUSE ante una reserva de memoria es la primer posición del bloque de datos reservado disponible**.

Aclaración: El tamaño máximo que cada hilo puede solicitar no está limitado por el tamaño de una página.

Memoria Virtual

Dado que trabajamos con direcciones lógicas, un proceso o hilo desconoce que comparte memoria con otros. Por lo tanto, cree que tiene **todo el espacio de direccionamiento** para él. Por obvias razones, cuando estén varios procesos compitiendo por la memoria, alguno llegará al punto donde **puede direccionar más memoria, pero no existen más espacios disponibles**.

Para solucionar este inconveniente, este módulo del trabajo práctico contará con **memoria virtual**[15], que permitirá a cada proceso acceder a un espacio de direccionamiento mayor al provisto por la memoria real, o en este caso la UPCM.

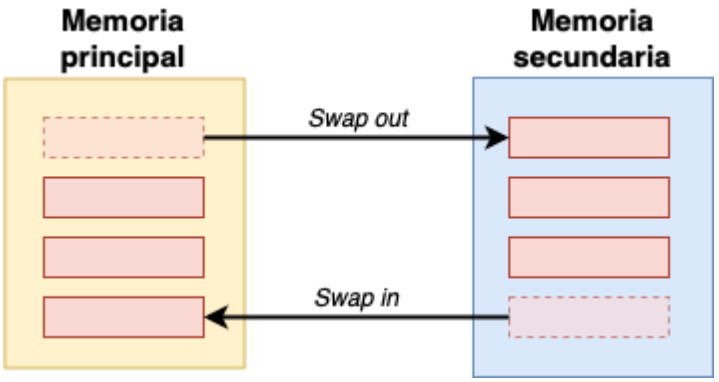
El algoritmo de reemplazo será de *asignación variable, alcance global*, utilizando ***clock modificado*** (también llamado de segunda oportunidad mejorado).[16]

Para la asignación de marcos se deberán recorrer los marcos disponibles siempre de menor a mayor, en función del número de marco.

Análogamente, eso implica que cada vez que se realice una operación sobre una página, MUSE validará si la misma está cargada en la memoria para poder ser utilizada. Si no se encuentra en memoria, se ejecuta una excepción conocida como ***page fault***, es decir, el faltante de una página.

En ese caso, MUSE irá a buscar la página faltante al área de swap y la cargará en la memoria principal, ejecutando el algoritmo de reemplazo de páginas en caso de ser necesario.

[15]Ver el Capítulo 9: Memoria Virtual del libro “Fundamentos de Sistemas Operativos”, Silbertchatz 7ma Edición.
[16]Ver el Capítulo 9: Sustitución de Páginas (9.4.5.3) del libro “Fundamentos de Sistemas Operativos”, Silbertchatz 7ma Edición.



La implementación de MUSE del área de swap será un único **archivo de swap** en el sistema de archivos. Diremos entonces, que MUSE deberá todo el tiempo tener conocimiento sobre las páginas que tiene cargadas en memoria y las que tiene en archivo de swap.

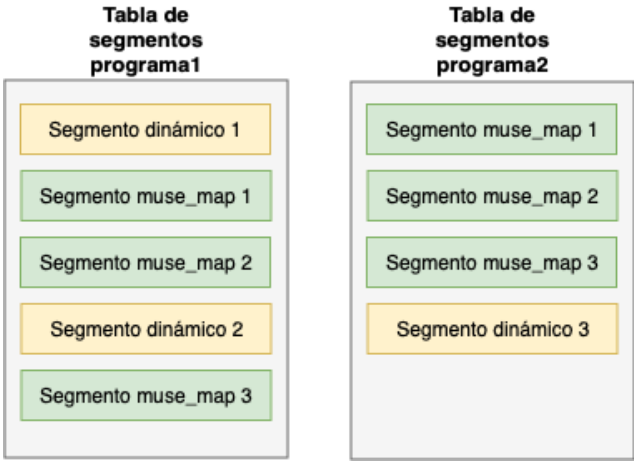
Dicho archivo de swap se encontrará **dividido** en espacios del tamaño de página para poder ser indexado. Este índice será utilizado para ubicar la página en caso de que la misma se encuentre en swap.

Memoria Compartida

Una funcionalidad más que tendrá nuestro módulo de memoria, es la capacidad de **compartir segmentos**. Para ello MUSE implementará un par de funciones que podrán reemplazar a mmap (y a un par de funciones de la misma familia): `muse_map`, `muse_sync` y `muse_unmap`.
Por simplicidad, esta implementación solamente recibirá un **path** en vez de un **descriptor de archivo**.

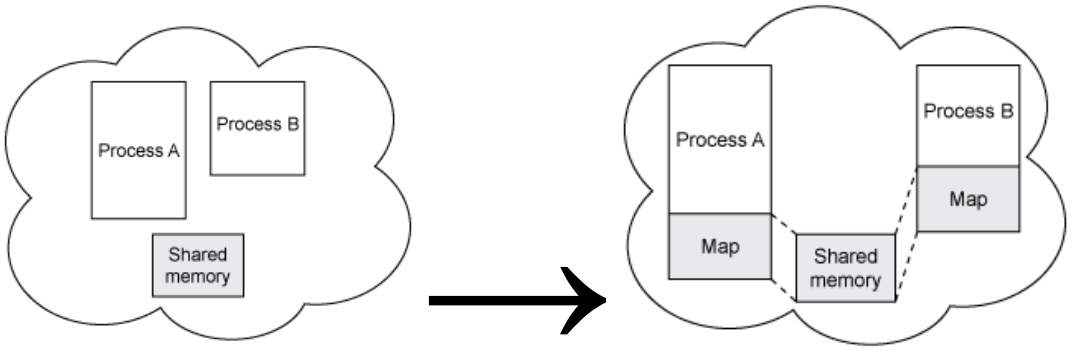
```
#include <unistd.h>                    ←→  #include “libmuse.h”
#include <sys/types.h>                 ←→  #include <stdio.h>
#include <sys/mman.h>                  ←→
#include <sys/stat.h>                  ←→
#include <fcntl.h>                     ←→
#include <stdio.h>                     ←→

void main() {                              void main() {
   size_t filesize = 16;                      size_t filesize = 16;
   int fd = open("hola.txt", O_RDONLY,  ←→    uint32_t map = muse_map(“hola.txt”,
0);                                     ←→ filesize, MAP_PRIVATE);
   void* map = mmap(NULL, filesize,     ←→
NULL, MAP_PRIVATE, fd, 0);                 ←→

   printf(map, filesize);                     printf(map, filesize);
                                           ←→  int x = 4;
   *(int*) map = 4;                        ←→  muse_cpy(map, &x, 4);
   msync(map, filesize, MS_SYNC);          ←→  muse_sync(map, filesize);

   munmap(map, filesize);                  ←→  muse_unmap(map);
   close(fd);                              ←→
}                                              }
```

Al tener porciones de memoria compartida mapeada a un archivo diferente al de swap, es necesario poder distinguirlas a nivel metadatos. Es por eso, que existirán 2 tipos de segmentos en la tabla de segmentos, los comunes (o dinámicos) y los compartidos (o "*mappeados*"):



Esta capacidad de compartir segmentos nos permitirá que dos o más **procesos diferentes puedan tener la misma porción de memoria**, siendo este un método de InterProcess Communication o *IPC*.

Ejemplificando un poco como funciona la implementación actual en linux, asumamos que tenemos dos procesos, A y B, que queremos comunicar mediante este IPC. Alguno de estos procesos tratará de reservar memoria compartida entre ellos, lo cual disparará la creación de **segmentos que compartan la misma tabla de páginas**:



Cuando el otro proceso solicite crear un nuevo segmento, el sistema operativo **reutilizará la tabla de paginas del mismo**.

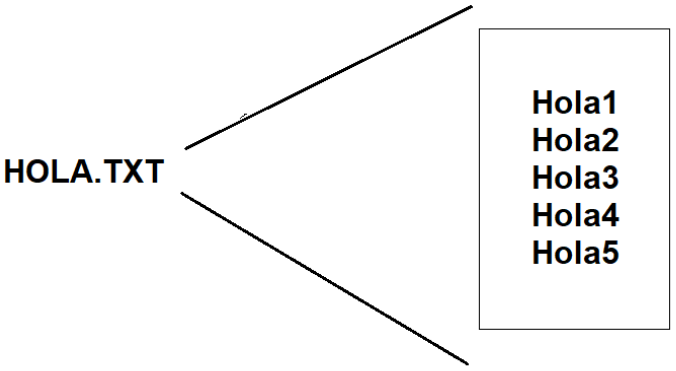
Este proceso de mapeo los sistemas como Linux (POSIX compliant) lo realizan mediante la funcionalidad que ya mencionamos, **mmap**.

Apalancándonos en el mismo mecanismo que permite el *swapping* de páginas, la funcionalidad de memoria compartida que proveerá MUSE (a través de sus funciones de *muse_map*) se realizará sobre

un **archivo compartido**[17], en vez del archivo de swap. Esta distinción deberá estar plasmada en la tabla de segmentos.

De la misma forma que el archivo de swap, cuando se mapea el archivo, no se dispondrá completamente en memoria, **sino que solamente las páginas que quieran ser accedidas**. Vamos a explicar esto en detalle con un ejemplo:

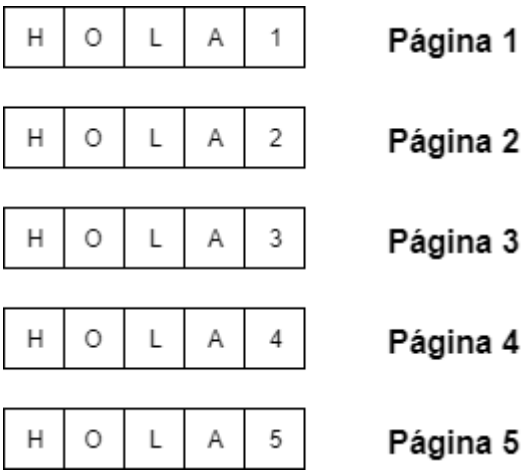
Tenemos el archivo "hola.txt" con el contenido como se ve en la imagen.



Para simplificar el ejemplo, no tengamos en cuenta caracteres especiales como \n (salto de línea), \0 (fin de cadena) o EOF (fin de archivo), y *digamos que en una página entran cinco caracteres*.

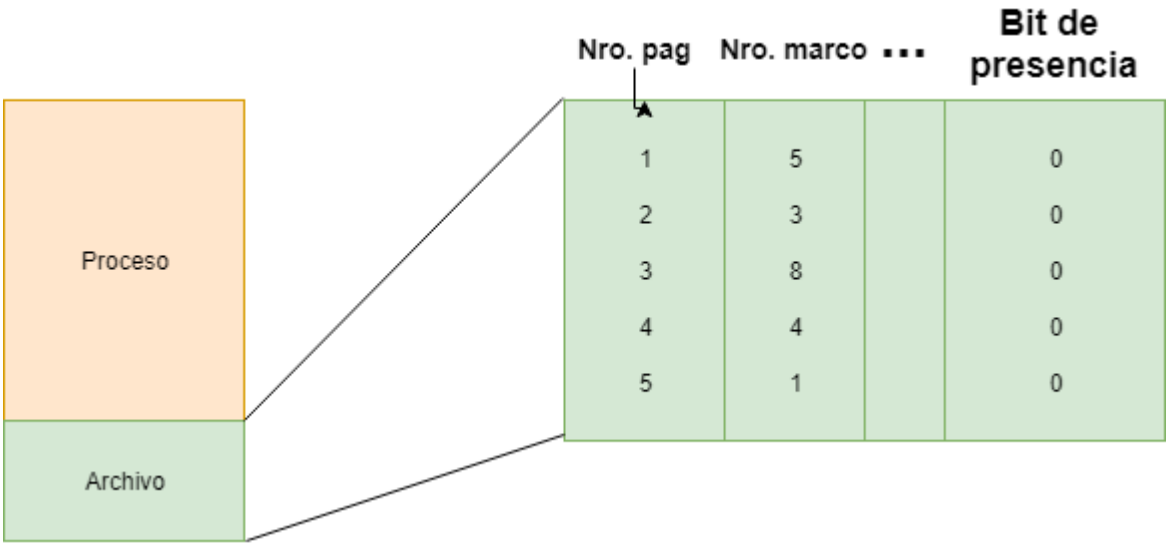
Al solicitar una operación de muse_map del archivo "hola.txt", la secuencia de eventos debería ser:

1. **"Separa" lógicamente a dicho archivo en páginas.**



2. Se anexa en la tabla de segmentos el segmento del archivo mapeado a memoria. La tabla de páginas de ese segmento tiene las cinco páginas, **todas ellas con el bit de presencia en 0.**

[17] El archivo estará dado como **parámetro** al momento de llamar a muse_map. Ver la interfaz propuesta en https://github.com/sisoputnfrba/libmuse/blob/master/src/libmuse.h

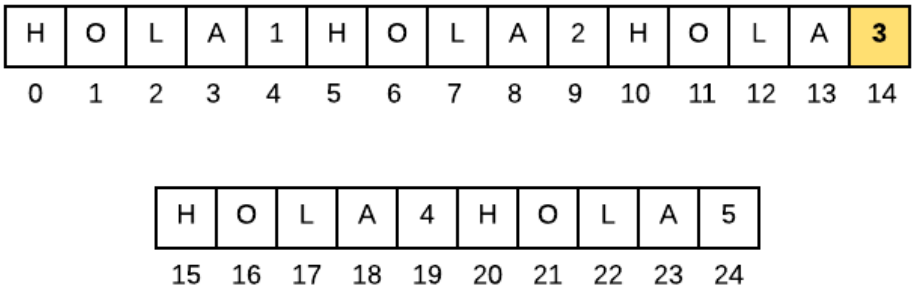


3. Cuando queramos leer el contenido, digamos de la página 1, se generará un *page fault* por esa página, y recién en ese instante se cargará en memoria para poder utilizarla, cambiando en el proceso el bit de presencia a 1.
4. Cuando modifico el contenido de una página, el bit de modificado de esta cambia a 1. Eventualmente cuando la página sea elegida como víctima del algoritmo de reemplazo ese cambió se verá reflejado en el archivo abierto mediante mmap. Si nunca se reemplazó la página, el cambio se hará cuando cierre el mmap.
5. Para cerrar el mmap, uso la llamada al sistema munmap.

Cuando *mapeamos* un archivo podemos manipular el contenido como si fuera un array. Siguiendo el ejemplo anterior, supongamos que quiero modificar el “hola3” por “hola0”. Nos quedaría un pseudocódigo más o menos así:

```
char* mi_mapeo = mmap(“hola.txt”);

mi_mapeo[14] = ‘0’
```

El 14 se debe a que como lo que fue mapeado a memoria lo veo como un array, accede a la posición que le corresponde a ese caracter, como se ve en la siguiente imagen:



IP

Tengamos en cuenta, que en C un caracter tiene el tamaño de un byte, por lo que suele ser utilizado como un tipo “alias” para el byte.

El comportamiento que se espera de esta funcionalidad es el mismo que el del mmap real con un archivo, con la diferencia que utilizaremos la ruta para abrir el archivo (en vez de un file descriptor). Tener en cuenta que también deberá diferenciar entre “muse_maps” **privados o compartidos**, mediante los flags.

Al momento que se libere el segmento compartido con “muse_unmap” se deberá tener en cuenta que si dos programas tienen un archivo abierto, **solo se debe liberar la tabla de páginas no haya segmentos direccionandola** (es decir, todos hicieron un “unmap”).

Archivo de Configuración[18]

Campo	Tipo	Descripción
LISTEN_PORT	[numérico]	Puerto TCP utilizado para recibir las conexiones de CPU y I/O
MEMORY_SIZE	[numérico]	Tamaño de la memoria en bytes
PAGE_SIZE	[numérico]	Tamaño de la página en bytes
SWAP_SIZE	[numérico]	Tamaño del swap en bytes. Será múltiplo del tamaño de página.

Ejemplo de Archivo de Configuración

```
LISTEN_PORT=5003
MEMORY_SIZE=4096
PAGE_SIZE=32
SWAP_SIZE=1024
```

[18] Queda a decisión del grupo el agregado de más parámetros al mismo.

Módulo de File System

Este módulo nos permitirá **implementar nuestro propio almacenamiento de archivos**, que almacene los datos de forma **centralizada**. A diferencia de los otros dos módulos, este módulo no poseerá una biblioteca, sino que vamos a utilizar directamente un Sistema de archivos (o File System).

Los File Systems son la forma que definen los Sistemas Operativos para poder gestionar los datos guardados en un medio. Si bien hay varios de miles de implementaciones de File Systems, todos suelen respetar una interfaz en común mediante **llamadas al sistema**.

Eso significa, que no importa la implementación del programa que use este módulo, sino **qué archivos está accediendo**:

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int num;
  FILE *file =
fopen("~/folder/file.txt","w");
  fprintf(file,"%d", 12);
  fclose(file);
  return 0;
}
```

⟷

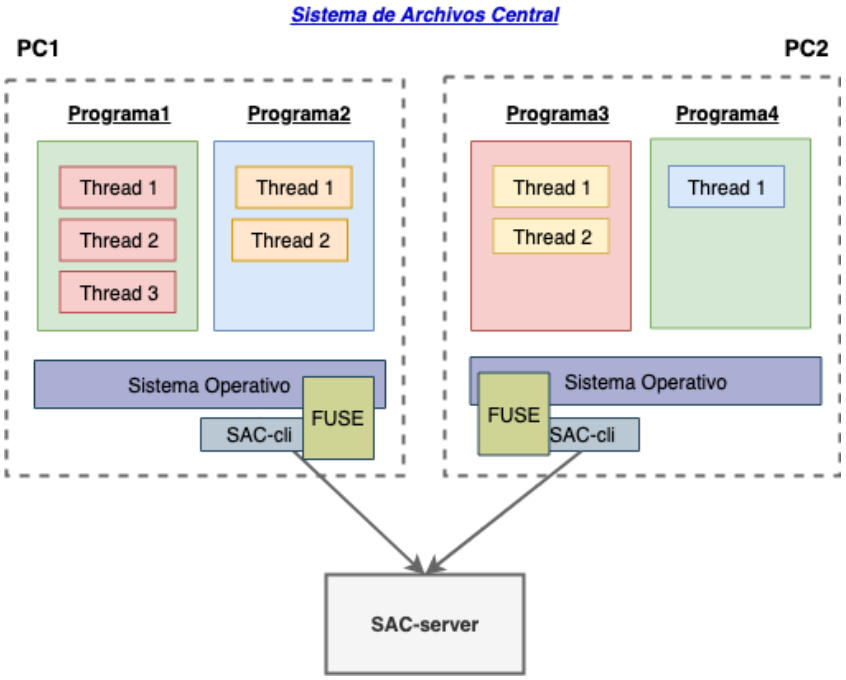
```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int num;
  FILE *file =
fopen("~/sac/file.txt","w");
  fprintf(file,"%d", 12);
  fclose(file);
  return 0;
}
```

El problema que tiene esta interfaz, es que las implementaciones de File Systems **se ejecutan en espacio de Kernel**, lo que provoca que cualquier error en la implementación de ese File System, **afecte al Sistema Operativo entero** (ni hablar que interactuar directamente con el SO suele ser bastante tedioso).

Tratando de solucionar un poco ese problema, utilizaremos un framework llamado **FUSE** o FileSystem in Userspace, que actúa como una suerte de "traductor" entre el Sistema Operativo y nuestra implementación de un File System, **la cual ejecuta en espacio de usuario**.

Es decir, cuando el SO se encuentre en la necesidad de guardar o leer información, este se lo comunicará a FUSE y buscará dentro de las operaciones que el programador definió, cómo hacerlo.

Entre los submódulos, será el **SAC-cli** el que implemente el framework FUSE; comunicándose con el proceso de almacenamiento central o **SAC-server**.



De esta forma, cada uno de los SAC-cli funcionará como un **adaptador entre las llamadas al sistema que hagan los programas que usen el módulo y el SAC-servidor**, redirigiendo a él todas las consultas referidas a los archivos, directorios y metadata que posea.

Por lo tanto, cada grupo deberá **implementar todas las funciones de FUSE que sean necesarias** para proveer un correcto funcionamiento del FileSystem.

¿Cómo funciona FUSE?

Cuando un proceso llama a una función de I/O (ejemplo, un ls en la consola), se termina traduciendo a una (o más) llamadas al sistema. Estas syscalls son capturadas por el Kernel y delegada al VFS (Virtual FileSystem).

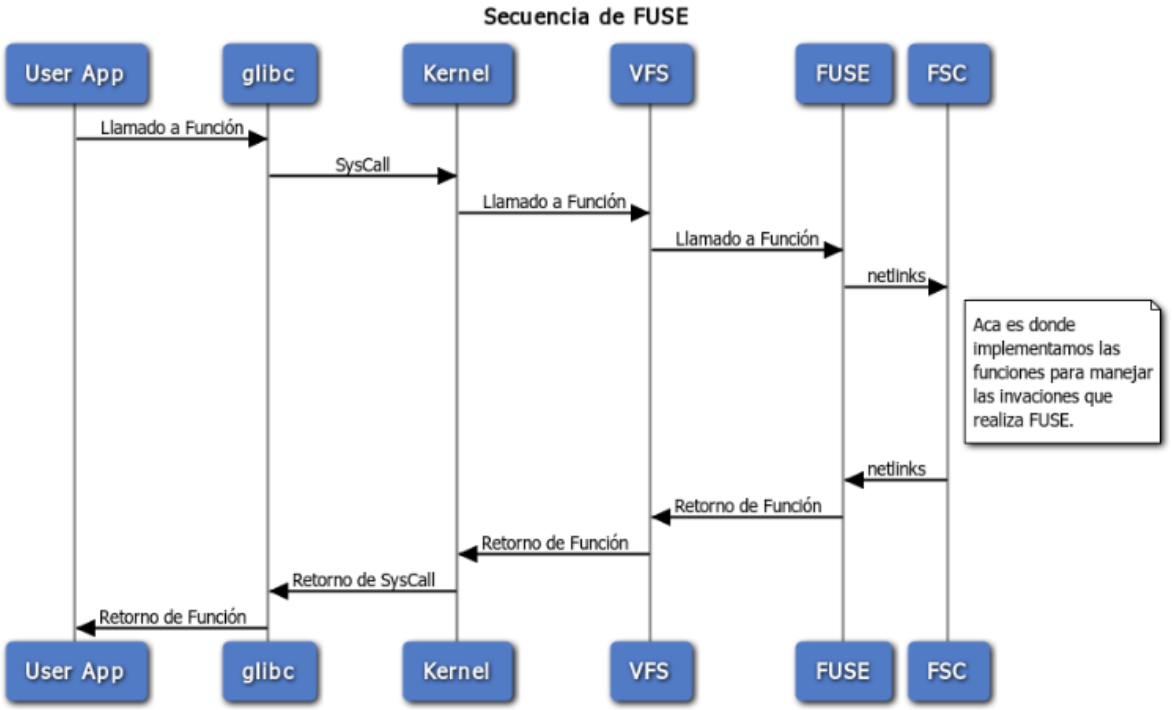
Si bien lo mencionamos antes sin nombrarlo, **el VFS es esa capa genérica** a todas las posibles implementaciones de FileSystem que el Kernel pueda tener.

El VFS se encarga de determinar (basándose en el path y los puntos de montaje) la implementación que le corresponde resolver la llamada al sistema. En este caso, la implementación de FUSE, quien, al recibir la orden, la delega al proceso que esté encargado de resolverla.

Internamente la biblioteca de FUSE trabaja con un mecanismo multi-thread, es decir, a cada operación se le asigna un nuevo thread de ejecución. Este mecanismo multi-thread puede ser deshabilitado pasándole como parámetro a FUSE ”-s”, opción que ***simplifica la tarea de debuguear*** (sin embargo, se remarca que el TP será evaluado en un entorno multithread).

Otro de los mecanismos internos que posee FUSE es una caché interna propia. En la mayoría de las condiciones estas traen beneficios en cuanto a evitar accesos de I/O. Para el caso del TP ***esta caché debe estar deshabilitada***.

Para anular la caché que posee la biblioteca de FUSE basta con pasar como parámetro la opción: ***-o direct_io.***



SAC-servidor

Este proceso gestionará un Filesystem almacenado en un archivo binario que será leído e interpretado como un árbol de directorios y sus archivos. El filesystem presentado será **central** a todos los programas que se ejecuten dentro de nuestro sistema, permitiéndole a cada uno realizar las siguientes operaciones:

- Crear, escribir, leer y borrar archivos.
- Crear y listar directorios y sus archivos.
- Eliminar directorios.
- Describir directorios y archivos.

SAC-Server debe poder realizar las distintas operaciones solicitadas por los programas/hilos en paralelo.

Características del file system:

Generales:

- Tamaño máximo de disco soportado: 16 TB
- Tamaño máximo de archivo 3.9GB
- Soporte de directorios y subdirectorios
- Tamaño máximo de nombre de archivo: 71 caracteres
- Cantidad máxima de archivos: 1024

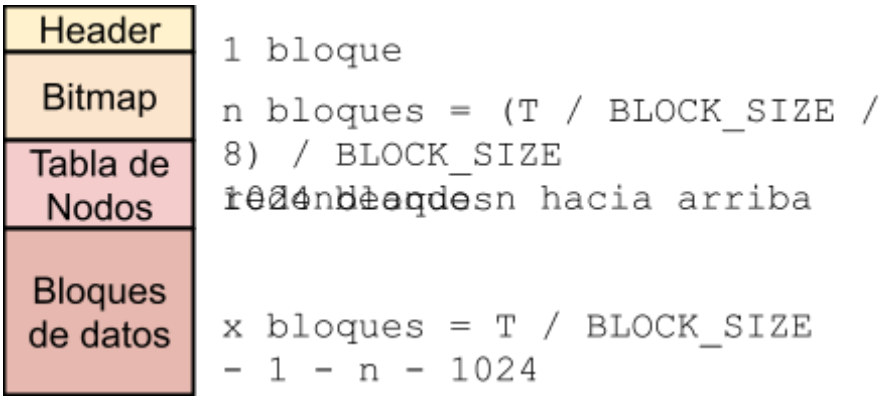
Técnicas:

- Tamaño de bloque: Fijo (4096 bytes)
- Tamaño de la tabla de nodos: 1024 bloques (4 MB)
- Tabla de bloques libres: Bitmap

Especificación del file system

Cada bloque en el sistema de archivos estará direccionado por un puntero de 4 bytes llamado *ptrGBloque* permitiendo así un máximo de 2^32 bloques.

Para un disco de tamaño T [bytes] y el BLOCK_SIZE de 4096 las estructuras se definen de esta manera:



Header

El encabezado del sistema de archivos estará almacenado en el primer bloque. Contará con los siguientes campos:

Campo	Tamaño	Valor
Identificador	3 bytes	“SAC”
Versión	4 bytes	1
Bloque de inicio del bitmap	*ptrGBloque* (4 bytes)	1
Tamaño del Bitmap (en bloques)	4 bytes	n
Relleno	4081 bytes	[no definido]

Bitmap

El Bitmap, también conocido como bitvector, es un **array de bits**[19] en el que cada bit identifica a un bloque a partir de su posición y nos permite conocer su estado, es decir, si se encuentra ocupado (1) o no (0). *Recordar que las implementaciones a nivel bit suelen ser susceptibles al Endianness* (o byte

[19] Ver el bitarray de las commons

ordering)[20]. *Por ejemplo*: un bitmap con este valor 000001010100 nos indicaría que los bloques 5, 7 y 9 están ocupados y los restantes libres (recordar que la primer ubicación es la 0).

Cuando el filesystem necesite localizar un bloque libre para la escritura de datos simplemente utilizará esta estructura para encontrarlo, marcándolo como ocupado una vez que lo utilizó.
Es **importante** recalcar que los bloques utilizados por las estructuras administrativas deben siempre estar marcados como utilizados.

Tabla de Nodos

La tabla de nodos de SAC es de tamaño fijo: un array de 1024 posiciones de estructuras de tipo GFile. Cada nodo consta de los siguientes campos:

Campo	Tamaño	Valor
Estado	1 byte	0: Borrado, 1: Ocupado, 2: Directorio
Nombre de Archivo	71 bytes	[Nombre del archivo]
Bloque padre	ptrGBloque	El número de bloque donde está almacenado el directorio padre o cero si está en el directorio raíz
Tamaño del archivo	4 bytes	(Máximo 4 GB)
Fecha de creación	8 bytes	Timestamp de creación del archivo
Fecha de modificación	8 bytes	Timestamp de la última modificación del archivo
Array de punteros a bloques indirectos simples	1000 x ptrGBloque	Cada posición del array contiene la dirección del bloque que almacena un array de 1024 direcciones de bloques de datos

Archivo de Configuración[21]

El proceso deberá poseer un archivo de configuración en una ubicación conocida donde se deberán especificar, al menos, los siguientes parámetros:

Campo	Tipo	Descripción
LISTEN_PORT	[numérico]	Puerto TCP utilizado para recibir las conexiones de CPU y I/O

[20] Ver Apéndice 10B del libro Organización y Arquitectura de Computadores, William Stallings; 7ta edición
[21] Queda a decisión del grupo el agregado de más parámetros al mismo.

Ejemplo de Archivo de Configuración

```
LISTEN_PORT=8003
```

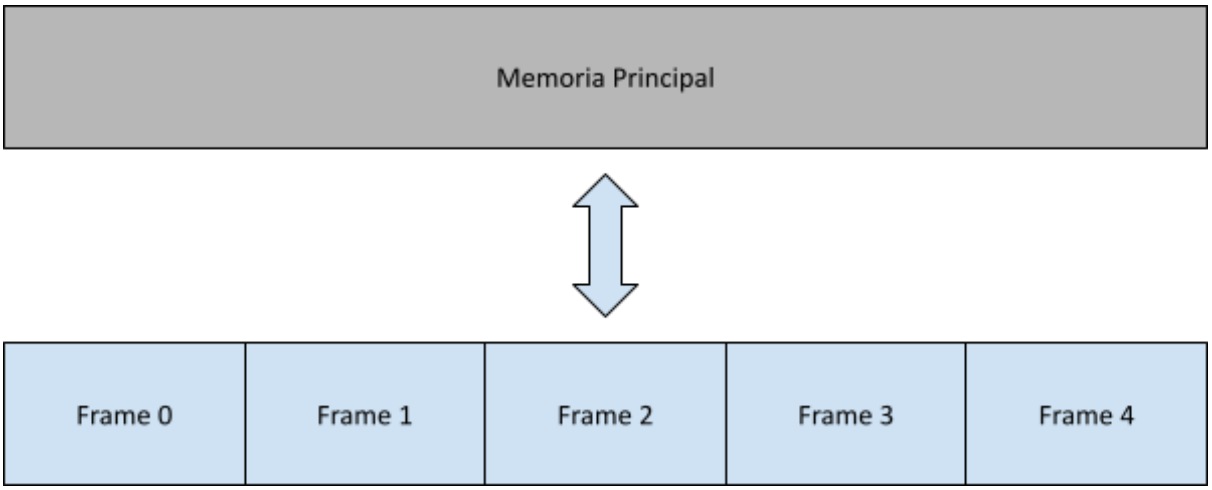
Anexo I - Memoria

En esta sección nos encargaremos de repasar y detallar el funcionamiento interno de la memoria a nivel físico como virtual. Para esto, comenzaremos definiendo algunas estructuras y a partir de ellas explicaremos el funcionamiento.

Estructuras

Memoria principal

Es una sección contigua de bytes que estará dividida en frames donde se almacenarán los datos de cada programa. El tamaño del mismo será definido por la propiedad de configuración "MEMORY_SIZE" y la cantidad de frames que contenga por la propiedad "PAGE_SIZE" (que será el tamaño de cada frame).



Para administrar esta sección ***utilizaremos mínimamente un bitmap*** el cual indique que frame se encuentra utilizado o no. Queda a disposición de cada grupo incorporar o no otra estructura administrativa para poder implementar el algoritmo clock modificado.

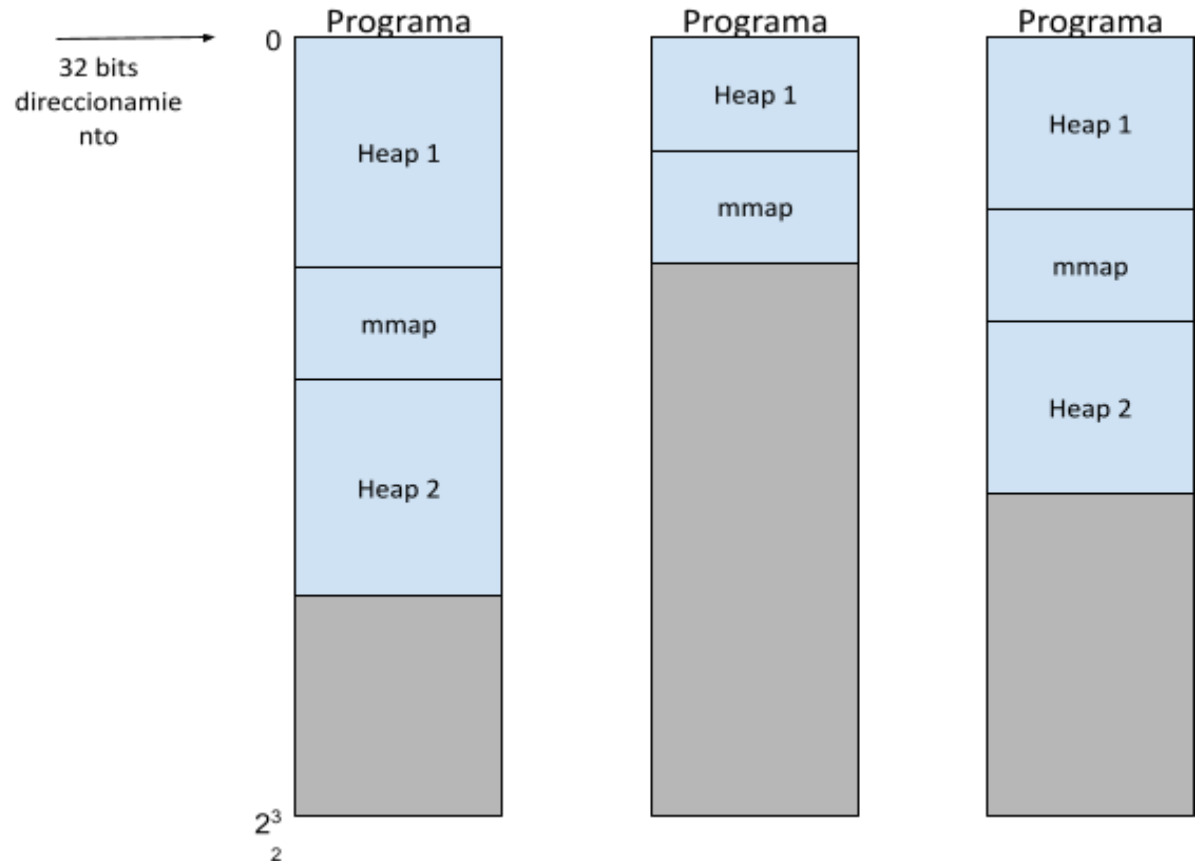
Espacio de direccionamiento virtual

Esta estructura administrativa será definida a nivel de cada programa y en ella se podrán ver todos los segmentos que posee cada uno en memoria. Cada espacio de direccionamiento podrá ser de hasta 32bits (espacio máximo de direccionamiento).

Los segmentos podrán ser de dos tipos:

1. Heap: Son aquellos segmentos creados por el programa a partir de la solicitud de diferentes muse_alloc.
2. mmap: Son aquellos segmentos que se crean a partir de la utilización de la función muse_map.

De esta manera un ejemplo podría estar representado de la siguiente manera:



Como se puede apreciar es correcto que tanto el Programa A, B y C al realizar el primer malloc el sistema le informe que su dirección virtual es “0” ya que cada espacio de dirección virtual es propio de cada programa.

En este punto podremos decir que ***un segmento es extendible siempre y cuando a continuación del mismo no exista otro segmento***. En el ejemplo mostrado, todos los segmentos de Heap 1 no son extensibles ya que a continuación existe un segmento de mmap pero si los segmentos Heap 2 ya que a continuación su espacio virtual está libre.

Cada segmento tendrá una tabla de páginas asociado, el cual indicará qué frames tiene utilizado cada uno y si los mismos se encuentran en memoria principal o en el área de swap. A partir de acá incluimos dos estructuras nuevas.

Área de SWAP

Esta estructura será un área que se utilizará dentro del filesystem, que permitirá extender la memoria principal cuando esta se llene. Al momento de detectar que es necesario enviar al área de Swap un frame, se utilizará el algoritmo de reemplazo Clock Modificado y se indicará en la tabla de páginas pertinente el bit de presencia en 0y de donde obtenerlo.

Tabla de Segmentos

Para los fines de este trabajo práctico, se desconoce el tamaño máximo de cada segmento. Esto, lejos de ser un problema, soluciona una limitación que tiene la segmentación paginada clásica: en este TP, es posible que un segmento utilice los 32 bits de direccionamiento, es decir, un segmento podría abarcar todo el espacio virtual de direcciones de un proceso.

Cada proceso tendrá una única tabla de segmentos con los campos:

- Base lógica (no confundir con la base física de segmentación pura)
- Tamaño
- Puntero a la tabla de páginas de este segmento.

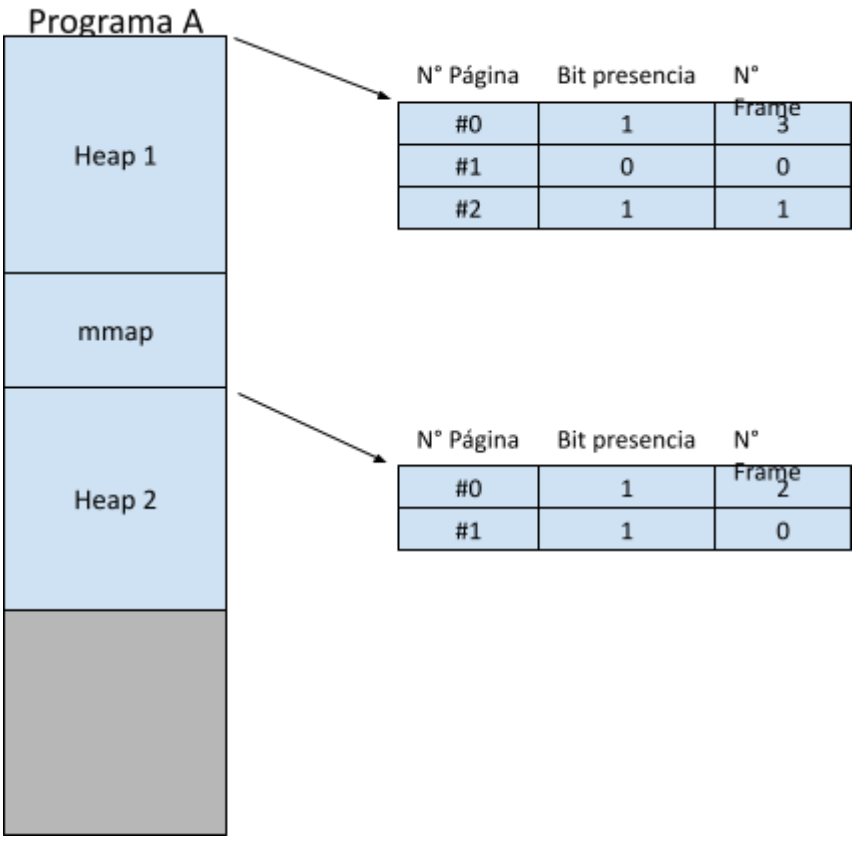
Ante la llegada de un acceso a memoria, vamos a chequear que esa dirección sea parte de alguno de nuestros segmentos. Si no lo fuera, se considerará acceso inválido. Si lo fuera, se accede a ese segmento y se busca la página correspondiente restando la base y partiendo la dirección en página y offset.

Tablas de Páginas

Cada segmento tendrá asociada una tabla de páginas en los cuales se indicará:

- Bit de presencia: Si la página se encuentra o no dentro de memoria principal
- Número de frame: En caso que se encuentre dentro de memoria indicará cual es el frame de memoria principal. En caso contrario indicará la posición de Swap.

Tanto para la tabla de páginas como la de segmentos, queda a decisión de cada grupo el agregado de otros campos. Estos agregados deberán ser justificados en el coloquio.

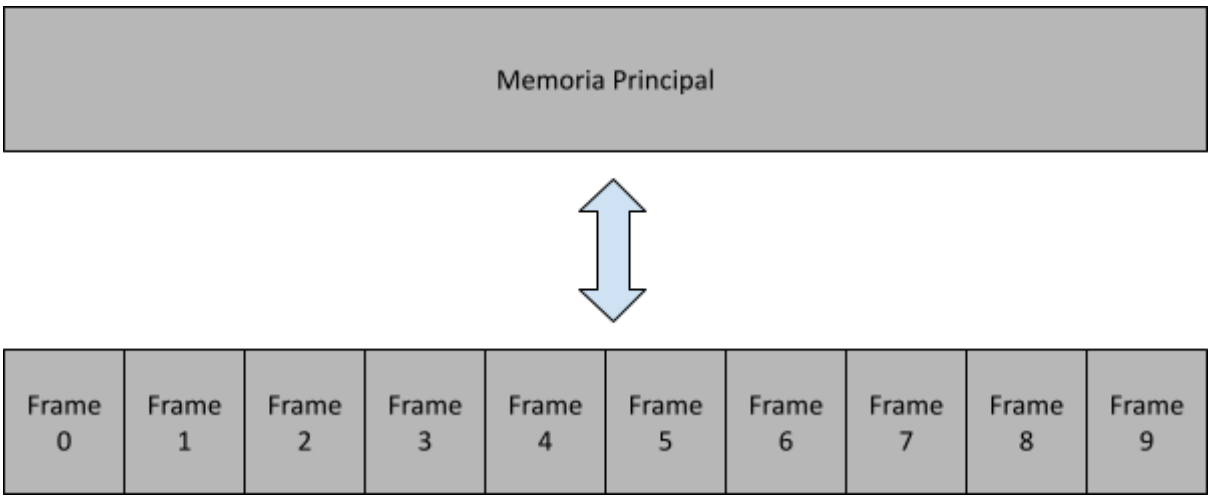


Queda a disposición de cada grupo poder agregar a dichas tablas los campos necesarios para administrar el clock modificado.

Funcionamiento

Para repasar todo lo dicho anteriormente realizaremos un ejemplo para comprenderlo de mejor manera. Tendremos dos Programas (Programa A y Programa B) los cuales irán realizando distintas operaciones contra memoria.

Inicialmente damos por sentado que nuestra memoria principal se encuentra vacía y tendrá disponibles 10 frames para todos nuestros programas y que cada uno de ellos tendrá 50 bytes (es decir, una memoria principal de 500 bytes).



Comenzaremos nuestro ejemplo con una solicitud del Programa A de muse_alloc(10). Al llegar dicha operación a nuestra memoria, primero buscaremos el espacio de direccionamiento virtual para el programa solicitado.

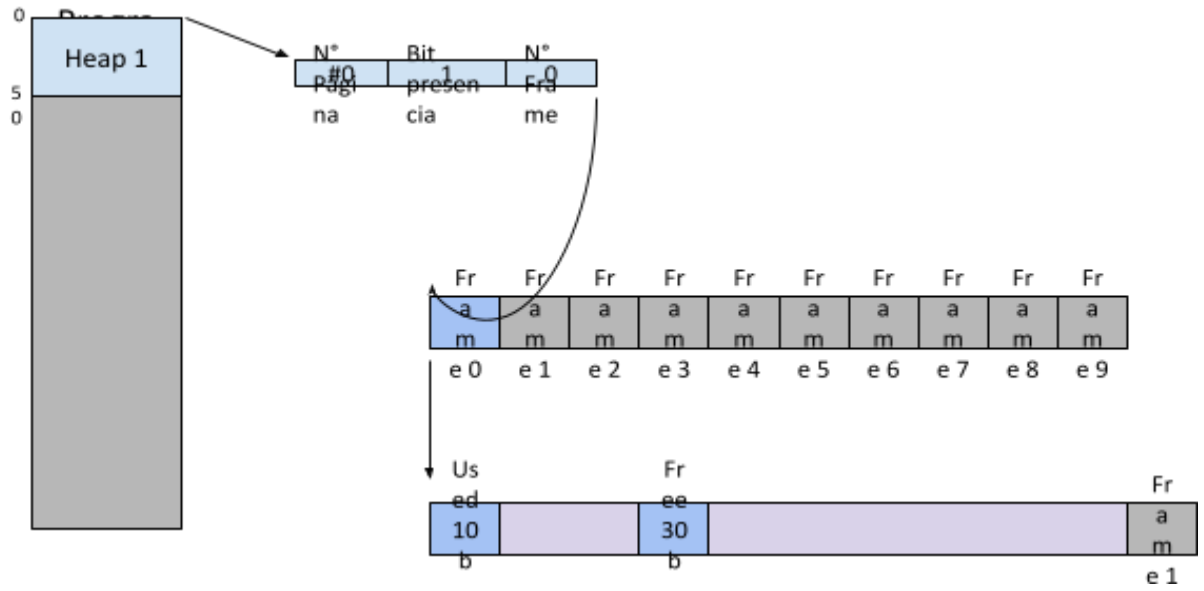
Eso se hará mediante el uso de un segmento de Heap en el cual pueda entrar nuestro dato (en caso de no existir ninguno, se deberá crearlo). De esa manera, analizaremos segmento a segmento de Heap, verificando si alguno de los Headers de metadata se encuentra con el valor free para incorporar el malloc. En caso de no encontrarlo, buscaremos si hay algún segmento de Heap que se pueda extender. Por último, en caso de no poder extender un segmento, deberá crear otro nuevo.

Volviendo a nuestro ejemplo, al detectar que no existe segmento de Heap se procederá a crear uno con el mínimo de frames necesarios para alocar los 10 bytes (un frame). Buscaremos en nuestro bitmap de memoria principal un frame que se encuentre libre asignando el “Frame 0”. Al realizar esto, crearemos un segmento de heap en nuestro espacio de direccionamiento virtual que vaya desde la dirección virtual 0 a la 50 (ya que reservamos espacio para un frame).

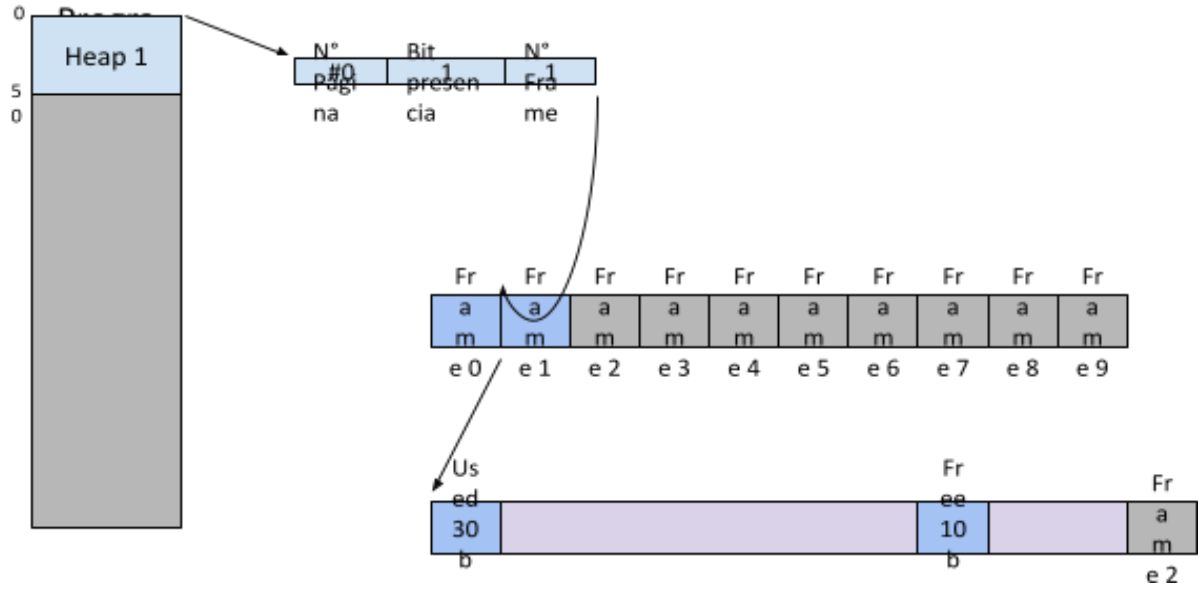
El segmento tendrá una tabla de páginas en la cual se refleja la página 0 de dicho segmento con el bit de presencia en 1 indicando que el número de frame asignado es el 0.

Por último, el Frame internamente poseerá una primera metadata para indicar que los primeros 10 bytes del mismo están siendo ocupados y otra metadata luego de estos 10 bytes para indicar que el restante se encuentra libre y disponible para usar. En este punto, nos damos cuenta que una primera solicitud de memoria incluye requerir el espacio solicitado más el tamaño de dos metadatas (5 bytes por cada uno).

El resultado del muse_malloc sera la posicion de memoria virtual donde comienzan los datos de dicha solicitud. En nuestro ejemplo sera la posicion 5 ya que los datos comienzan en el byte 5 de la primera página del segmento.



A continuación suponemos que nos llegará una solicitud del Programa B de muse_alloc(30). En este punto haremos homogéneamente lo mismo del paso anterior. Aquí asignaremos el Frame1 y nos quedará su estructura administrativa de la siguiente manera.



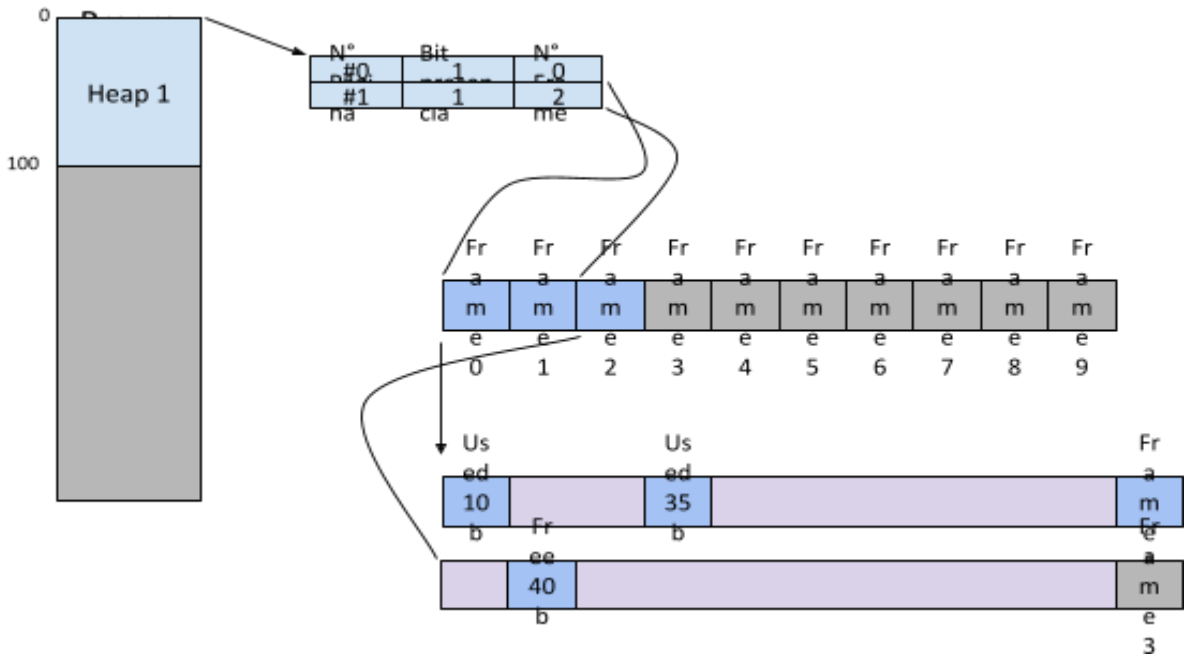
Suponiendo que a continuación llega una nueva solicitud de muse_alloc(35) del mismo Programa A, buscaremos dentro del espacio de direccionamiento de dicho programa un segmento de Heap que nos permite almacenar dichos 35 bytes. En este punto tenemos que tener en cuenta que para almacenar dichos 35 bytes necesitaremos a su vez minimamente 5 bytes más para guardar el siguiente metadata a dichos datos por lo que necesitaremos un total de 40 bytes.

Para esto, recorreremos dichos segmentos recorriendo las páginas de los mismos buscando aquella metadata que se encuentre como free y el tamaño disponible. En este punto descubriremos que no tenemos ningún segmento que califique así por lo que verificaremos si podemos extender un

segmento de Heap. En nuestro ejemplo al solo tener el segmento Heap 1 verificaremos que podemos extenderlo por lo que buscaremos dentro de nuestro bitmap un nuevo frame para asignarle (Frame 2). De esta manera, utilizaremos los 30 bytes del Frame 0 y los primeros 5 bytes del Frame 2 para los datos solicitados y los 5 bytes siguientes para la metadata requerida.

El resultado de dicho malloc será la posición de memoria virtual donde comienzan los datos de dicho malloc. En nuestro ejemplo será la posición 20 ya que los datos comienzan a partir del byte 20 de la primera página del segmento.

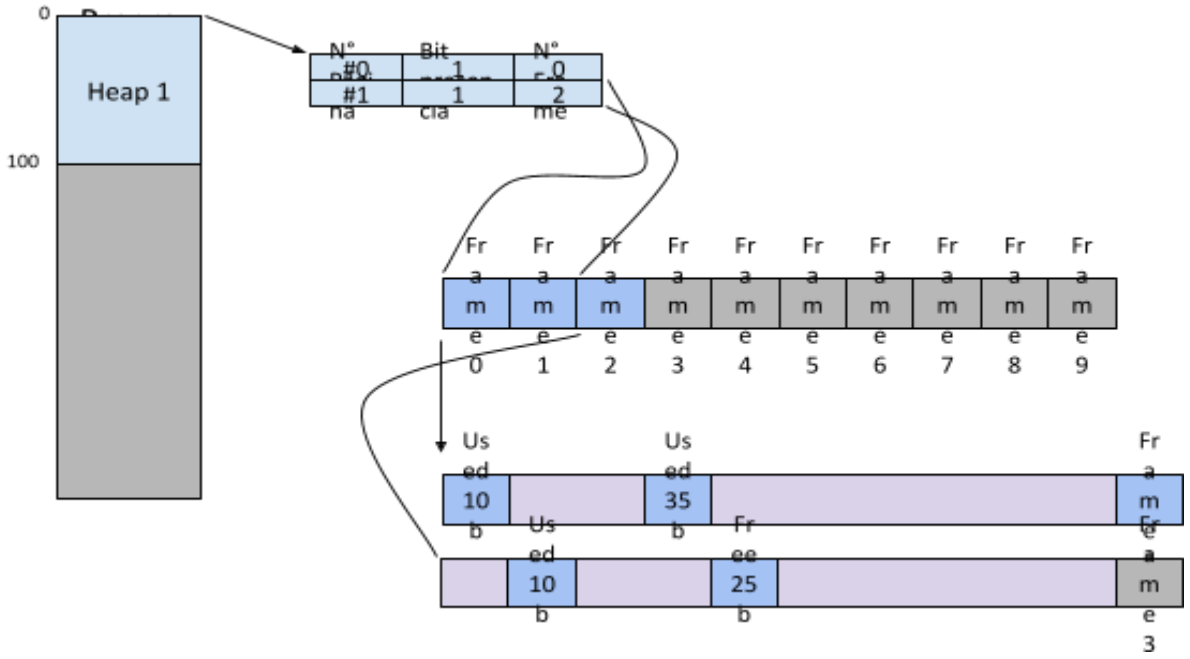
no



Continuando con el mismo Programa, realizaremos un muse_alloc(10). En este caso se podrá guardar dicho dato dentro del frame recién reservado. El resultado de dicho malloc será la posición 60 ya que los datos de dicho malloc comenzarán en el byte 10 de la segunda página de dicho segmento.

Para calcular el mismo, tomamos la base del segmento (en nuestro caso se encuentra en la posición virutal 0) y le sumamos el desplazamiento de la cantidad de páginas previas más el desplazamiento dentro de la página en la cual se encuentra.

De esta manera, al intentar leer dicho datos se solicitará un muse_get(60, 10). Dónde 60 será la dirección virtual y 10 es el tamaño de lo que se requiere leer. Para realizar una operación de lectura y escritura deberemos identificar en base a la dirección virtual a qué segmento nos referimos y una vez detectado tendremos que calcular a partir del desplazamiento dentro del mismo cual es la página del segmento que se requiere y por último el desplazamiento dentro del frame.



Para finalizar con el Programa A realizaremos un muse_free(5), donde 5 indicará la posición virtual 5. En este punto deberemos realizar como en el punto anterior identificar el segmento en cuestión (que en nuestro caso es el Heap 1 ya que se encuentra entre las posiciones 0 y 100). Una vez identificado, a partir del desplazamiento (posición solicitada - base) calcularemos el desplazamiento dentro del mismo obteniendo la página que se requiere.

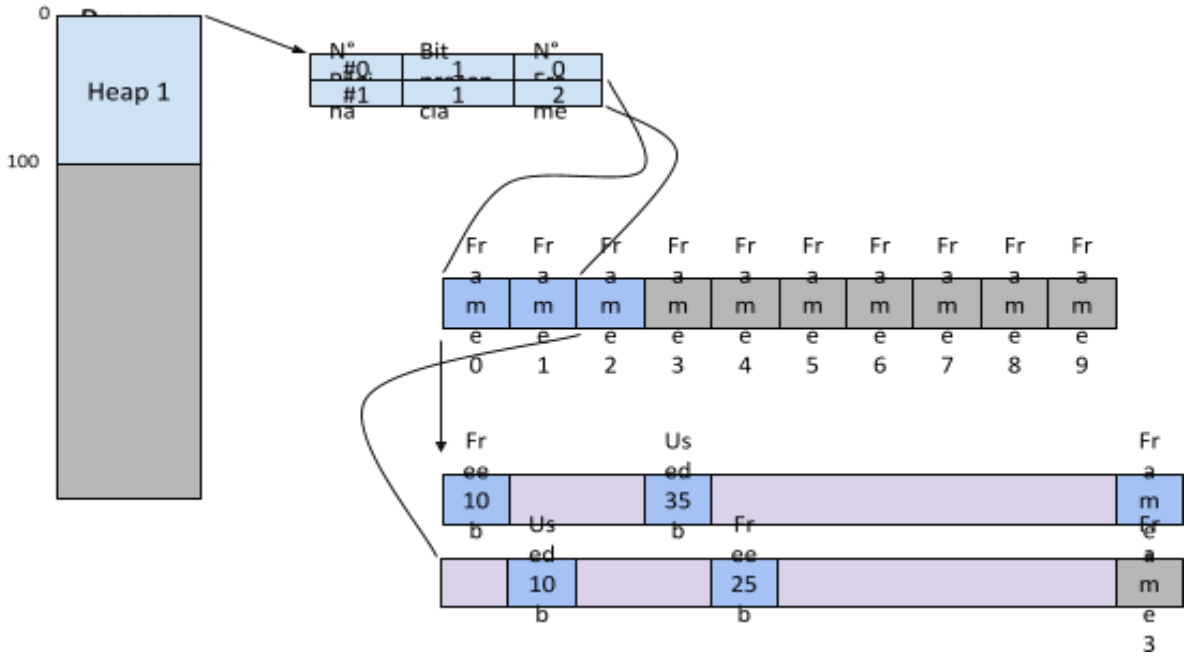
En nuestro caso es: (posición solicitada - base) / tamaño de frame => (5 - 0)/50 = 0.

Esto nos indica que es el numero de pagina 0 y dentro del mismo vamos a tener un desplazamiento de 5. Para calcular esto hicimos:

(posición solicitada - base) - (N° pagina * 50) => (5 - 0) - (0 * 50) = 5

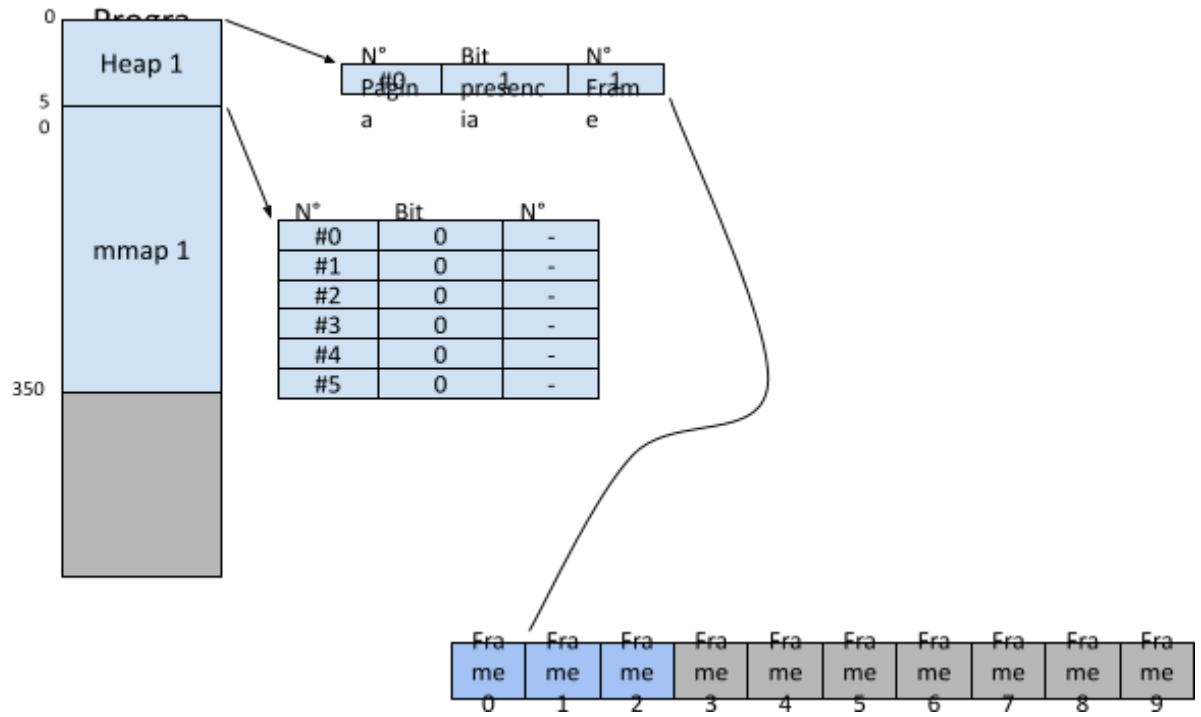
Una vez obtenido dicho valor, deberemos para resolver el free, cambiar la metadata indicando que dicha sección se encuentra libre. ***Se debe tener en cuenta que si la porción anterior o siguiente a dicho dato también se encuentra libre se deberán unificar en un único sector***.

Nuestro resultado final quedará así:



Volviendo a nuestro Programa B. Ahora supongamos que vamos a hacer un muse_map('file1.txt', 265, 0). De esta manera requerimos mapear del archivo file1.txt los primeros 265 bytes. Para esto, generamos un nuevo segmento de mmap con la cantidad de páginas necesarias para alocar 265 bytes (6 páginas). En la tabla de páginas todas las entradas inicialmente tendrán el bit de presencia en 0, puesto que todavía no se cargaron en memoria principal (se cargarán recién al querer accederlas).

Por último, el resultado del muse_map será la posición de memoria virtual donde comienza el segmento. En nuestro ejemplo sera la posicion 50.



En caso que se desee leer el byte 65 de dicho segmento se deberá solicitar el muse_get(115, 1) ya que la base virtual de dicho segmento es la 50.

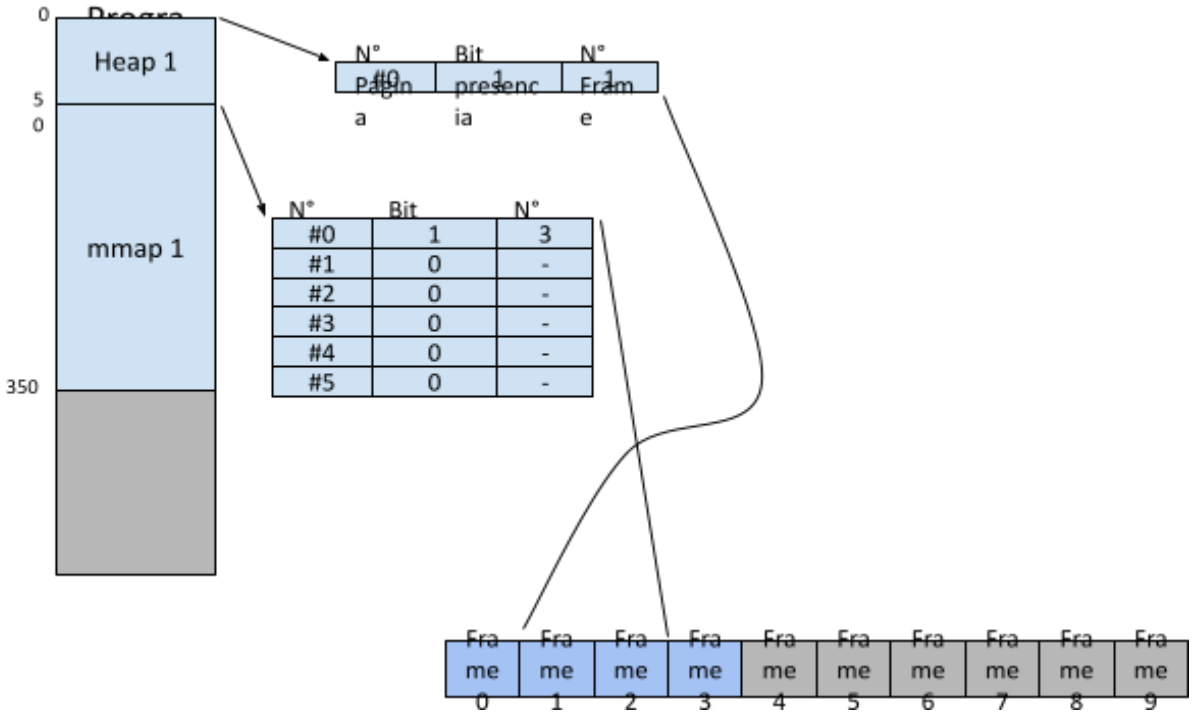
Al realizar esta operación, memoria detectará que la posición 115 se encuentra dentro del segmento mmap 1 y calculará utilizando las funciones anteriores cual es el numero de pagina de dicho segmento a requerir.

(posición solicitada - base) / tamaño de frame => (115 - 50)/50 = 1

Y el desplazamiento dentro de la página:

(posición solicitada - base) - (N° página * 50) => (115 - 50) - (1 * 50) = 15

De esta manera sabremos que tendremos que leer a partir de la posición 15 de la página 1 de dicho segmento. Encontramos que esa pagina tiene el bit de presencia en 0, por lo que no está cargada en memoria principal, así que se dispara un page fault. Esto implica tener que ir a reservar un frame (o tantos como fuesen necesarios), volcar en este el contenido del archivo y actualizar la tabla de páginas (para indicar el frame correspondiente y setear el bit de presencia). En nuestro ejemplo se reserva el frame 3. Finalmente tenemos que leer a partir del byte 15 de este frame.
Cabe destacar que en caso de no haber suficientes frames disponibles se deberá correr el algoritmo de reemplazo.


0
Heap 1
5
0
mmap 1
350
N°
Bit
N°
#0
1
3
#1
0
-
#2
0
-
#3
0
-
#4
0
-
#5
0
-
Frame 0
Frame 1
Frame 2
Frame 3
Frame 4
Frame 5
Frame 6
Frame 7
Frame 8
Frame 9

Anexo II - Directorio

La mayoría de los Sistemas de Archivos suelen poseer el concepto de directorio, para poder tener múltiples archivos con el mismo nombre, agrupar archivos y, particularmente, porque es más fácil recordar algo como "/home/" que "0x475561234". Si bien las implementaciones tradicionales son vistas en la parte teórica de la materia, el motivo de este anexo es demostrar en una forma explícita cómo la estructura de directorios funciona en el SAC-FS.

Nodos y bloques

Los **nodos** son aquellos que almacenan la metadata de los archivos, y **los bloques** aquellos que almacenan el contenido de los mismos. SAC-FS define que un nodo está asociado a ninguno, uno o un conjunto de bloques de datos. En implementaciones como ext2, los nodos de directorios suelen estar asociados a un bloque donde se contienen los nombres de sus directorios hijos y los bloques donde encontrarlos. Otros filesystems, como el ext3, utilizan un árbol de hashes (coloquialmente llamado Htree).

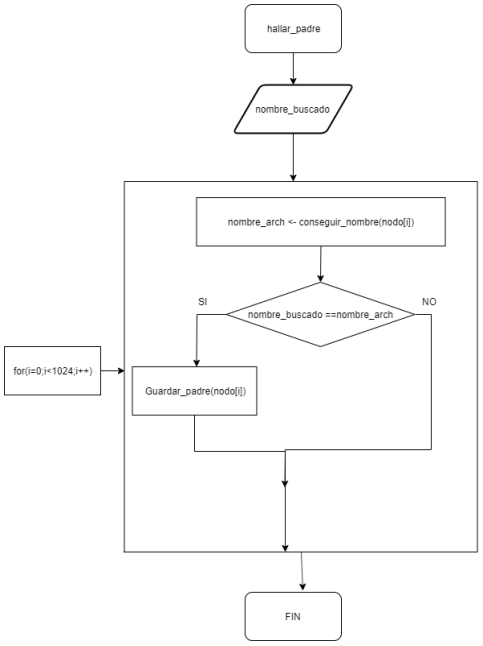
Como ellos, SAC-FS tiene la información de "quién es el padre de quien", usando los campos del **bloque padre** y el **nombre del archivo.** Para SAC-FS no es necesario crear bloques con entradas de directorio, sino que las mismas están implícitas en los nodos.

¿Bloque padre?

La lógica detrás de la existencia de esos campos es, precisamente, para poder determinar el camino en el árbol de directorios. Cuando llega una solicitud para un archivo en determinada ruta lo que hace SAC es buscar en toda la tabla de nodos el nombre de archivo que coincida con el último elemento de la ruta. Una vez determina todos los posibles candidatos, accede al padre mediante el puntero al bloque y repite el procedimiento, fijándose si la última entrada de la ruta (que ahora le suprimimos la anterior porque ya coincidió) existe, así hasta llegar a un 0 que determine que dicha entrada existe.

El algoritmo

A muy alto nivel, el algoritmo para la búsqueda del padre es la siguiente



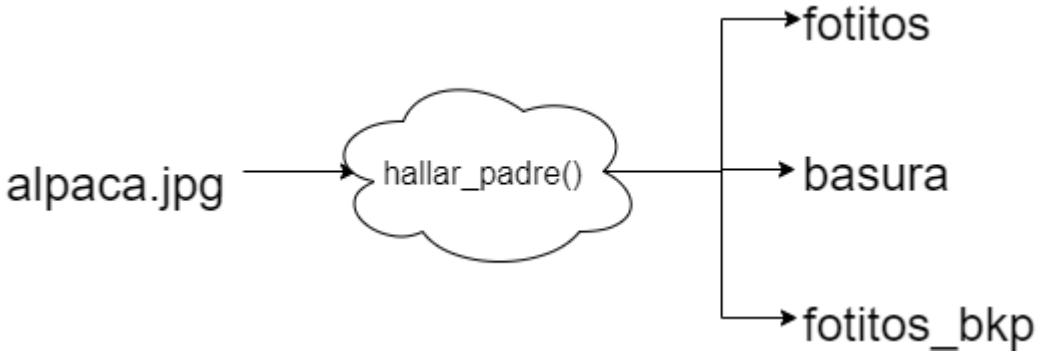
Para explicar en detalle el funcionamiento del algoritmo de directorios, vamos a hacerlo con un ejemplo que nos interesa a todos: **alpacas.**

Supongamos que nos llega una solicitud de lectura sobre la siguiente ruta:

`/facu/TPs/fotitos/alpaca.jpg`

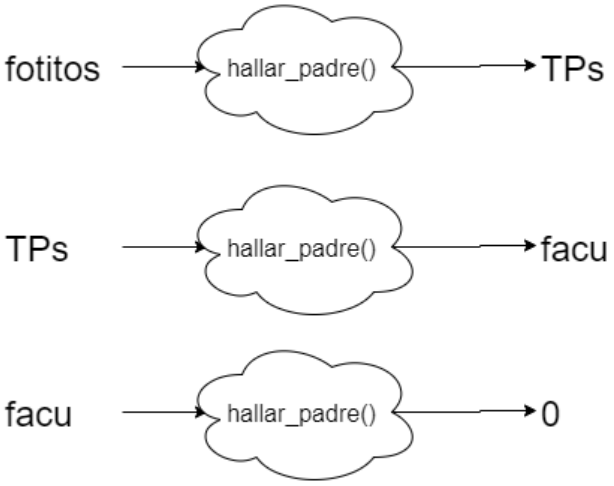
El procedimiento debería ser el siguiente:

- Obtener los distintos archivos que conforman la ruta : **[facu, TPs, fotitos, alpaca.jpg]**
- La búsqueda se realiza de derecha a izquierda, por lo que entonces busco todos los padres de los nodos cuyo nombre de archivo sea **alpaca.jpg**
- hallar_padre("alpaca.jpg") me devolvió tres bloques padre, ya que yo tenía tres "alpaca.jpg" en mi FS, pero como la ruta completa solo puede ser una **eventualmente los iré descartando ya que no van a coincidir todos nombres de archivos de los padres**. En este caso, afortunadamente, hay un solo padre cuyo nombre de archivo sea **fotitos.**



- itero nuevamente, ahora buscando los padres de "fotitos", luego los padres de TPs, y por últimos los de "facu"

- Finalmente, cuando busque el padre de “facu”, este me devolverá 0 y daré por finalizada la búsqueda



.

- Si en algún momento no logré llegar a “/”, significa que el directorio solicitado no existe y por lo tanto dicho archivo no existe.

Finalmente, hallamos nuestra foto y podemos visualizarla

SAC-Tools

Para simplificar la tarea de testing y hacer más transparentes las entregas, la cátedra provee las SAC-Tools, un script que permite a los alumnos crear un File System de SAC ya formateado. Este será el mismo que utilizaremos para generar los que utilizaremos durante las entregas. Las mismas se pueden encontrar en el siguiente repositorio.

Descripción de las entregas

Hito 1: Conexión Inicial

Fecha: 14/09/2019

Objetivos:

- ★ Familiarizarse con Linux y su consola, el entorno de desarrollo y el repositorio.
- ★ Aplicar las Commons Libraries, principalmente las funciones para listas, archivos de conf y logs
- ★ Definir el Protocolo de Comunicación
- ★ Empezar a investigar los ejemplos de FUSE

Implementación mínima:

- ★ Una biblioteca muy básica que permita enviar strings hasta un proceso que actúe de servidor.

Lectura recomendada:

- Tutorial de "Cómo arrancar" de la materia: http://faq.utnso.com/arrancar
- Beej Guide to Network Programming - http://faq.utnso.com/beej https://beej.us/guide/bgnet/
- SO UTN FRBA Commons Libraries - https://github.com/sisoputnfrba/so-commons-library
- Guía de Punteros en C - http://faq.utnso.com/punteros

Hito 2: Avance del Grupo

Fecha: 12/10/2019

Objetivos:

- ★ **Módulo de Planificación**: Permitir solamente planificar de forma FIFO un conjunto de ULTs de un mismo proceso (sin estado Blocked).
- ★ **Módulo de Memoria**: Implementación completa de la biblioteca libMUSE comunicándose con un MUSE de prueba (devolviendo respuestas fijas).
- ★ **Módulo de FS**: Implementación de FUSE a un programa de ejemplo.

Lectura recomendada:

- Sistemas Operativos, Silberschatz, Galvin - Capítulo 3: Procesos y Capítulo 4: Hilos
- Sistemas Operativos, Silberschatz, Galvin 7ma Ed. - Capítulo 8: Memoria Principal
- Ejemplo de Implementación de FUSE - https://github.com/sisoputnfrba/so-fuse_example

Hito 3: Checkpoint Presencial en el Laboratorio

Fecha: 02/11/2019

Objetivos:

- ★ **Módulo de Planificación**: Implementación de Semáforos junto al estado Blocked y estado NEW mediante el grado de multiprogramación.
- ★ **Módulo de Memoria**: Implementación de un sistema de Segmentación Paginada sin Memoria Virtual.
- ★ **Módulo de FS**: SAC-cli capaz de comunicarse con un SAC-servidor sencillo que solo lea del FileSystem Local.

Lectura recomendada:

- Sistemas Operativos, Silberschatz, Galvin 7ma Ed. - Capítulo 6: Sincronización de Procesos
- Sistemas Operativos, Silberschatz, Galvin 7ma Ed. - Capítulo 8: Memoria Principal
- Sistemas Operativos, Silberschatz, Galvin 7ma Ed. - Capítulo 10: Interfaz del Sistema de Archivos

Hito 4: Avance del Grupo

Fechas: 16/11/2019

Objetivos:

- ★ **Módulo de Planificación**: Implementación del algoritmo SJF, desarrollo de métricas.
- ★ **Módulo de Memoria**: Implementación de Memoria Virtual y Memoria Compartida en desarrollo.
- ★ **Módulo de FS**: Avances sobre la implementación del FileSystem de SAC.

Implementación mínima de módulos en desarrollo:

- ★ **Módulo de Memoria**: Permitir swap de páginas a varios archivos para habilitar memoria compartida.
- ★ **Módulo de FS**: Capacidad de leer de un bitmap e identificar la tabla de nodos.

Lectura recomendada:

- Sistemas Operativos, Silberschatz, Galvin 7ma Ed. - Capítulo 9: Memoria Virtual
- Sistemas Operativos, Silberschatz, Galvin 7ma Ed. - Capítulo 11: Implementación de Sistemas de Archivos

Hito 5: Entregas Finales

Fechas: 30/11/2019 - 7/12/2019 - 21/12/2019

Objetivos:

- ★ Probar el TP en un entorno distribuido
- ★ Realizar pruebas intensivas
- ★ Finalizar el desarrollo de todos los procesos
- ★ Todos los componentes del TP ejecutan los requerimientos de forma integral, bajo escenarios de stress.

Lectura recomendada:

- Guías de Debugging del Blog utnso.com - https://www.utnso.com/recursos/guias/
- MarioBash: Tutorial para aprender a usar la consola - http://faq.utnso.com/mariobash
- Tutoral de como desplegar un proyecto - https://github.com/fedebonisconti/so-deploy